NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEEP LEARNING FOR MEDIA ANALYSIS IN DEFENSE SCENARIOS—AN EVALUATION OF AN OPEN-SOURCE FRAMEWORK FOR OBJECT DETECTION IN INTELLIGENCE-RELATED IMAGE SETS

by

Taylor H. Paul

June 2017

Thesis Advisor:	Mathias Kölsch
Second Reader:	Michael McCarrin

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2017	Master's Thesis 07-06-2015 to 06-17-2017

4. TITLE AND SUBTITLE

DEEP LEARNING FOR MEDIA ANALYSIS IN DEFENSE SCENARIOS—AN EVALUATION OF AN OPEN-SOURCE FRAMEWORK FOR OBJECT DETECTION IN INTELLIGENCE-RELATED IMAGE SETS

5. FUNDING NUMBERS

6. AUTHOR(S)

Taylor H. Paul

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Department of Defense struggles to develop and maintain cutting-edge software through the Defense Acquisition System. The pace of improvements in machine learning algorithms and software suggests the organization will fail to rapidly develop systems incorporating the latest innovations to meet its intelligence-related media analysis needs. In contrast, the trend of industry and academia releasing algorithms and software under permissive licenses bestows defense organizations with an opportunity. These groups can potentially overcome resource shortfalls and long acquisition timelines by implementing machine learning solutions with open-source software. We test this hypothesis by employing an open-source software library to evaluate publicly available deep learning algorithms on three prior defense-related datasets. We then compare performance of deep convolutional neural networks to past methods for detecting AK-47s, ships, and screenshots in images. Applying proven algorithms through the software framework, we test three object detectors that exceed or match classification performance for all three experiments in a third of the development time available to designers of the previous algorithms. We relate these tests to defense scenarios in order to provide a logical argument and empirical measure of the utility of open-source machine learning frameworks to meet the Department of Defense's intelligence-related media analysis needs.

14. SUBJECT TERMS	15. NUMBER OF PAGES
deep learning, machine learning, object detection, computer forensics, open source, TensorFlow	135
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DEEP LEARNING FOR MEDIA ANALYSIS IN DEFENSE SCENARIOS—AN EVALUATION OF AN OPEN-SOURCE FRAMEWORK FOR OBJECT DETECTION IN INTELLIGENCE-RELATED IMAGE SETS

Taylor H. Paul
Captain, United States Marine Corps
B.S., United States Naval Academy, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2017

Approved by: Mathias Kölsch
Thesis Advisor

Michael McCarrin
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Department of Defense struggles to develop and maintain cutting-edge software through the Defense Acquisition System. The pace of improvements in machine learning algorithms and software suggests the organization will fail to rapidly develop systems incorporating the latest innovations to meet its intelligence-related media analysis needs. In contrast, the trend of industry and academia releasing algorithms and software under permissive licenses bestows defense organizations with an opportunity. These groups can potentially overcome resource shortfalls and long acquisition timelines by implementing machine learning solutions with open-source software. We test this hypothesis by employing an open-source software library to evaluate publicly available deep learning algorithms on three prior defense-related datasets. We then compare performance of deep convolutional neural networks to past methods for detecting AK-47s, ships, and screenshots in images. Applying proven algorithms through the software framework, we test three object detectors that exceed or match classification performance for all three experiments in a third of the development time available to designers of the previous algorithms. We relate these tests to defense scenarios in order to provide a logical argument and empirical measure of the utility of open-source machine learning frameworks to meet the Department of Defense's intelligence-related media analysis needs.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

List of Acronyms and Abbreviations

ANN artificial neural network

API application program interface

AR assault rifle, assault gun

BOW bag-of-words

CIO Chief Information Officer

CNN convolutional neural network

CNTK Microsoft Cognitive Toolkit

COCO Common Objects in Context

DAS Defense Acquisition System

DOD Department of Defense

DODD Department of Defense Directive

DPM deformable parts model

FLOP floating-point operation

FPR false positive rate

GPU graphics processing unit

HOG histogram of oriented gradients

ILSVRC ImageNet Large Scale Visual Recognition Challenge

ISR intelligence, surveillance, and reconnaissance

MLP multilayer perceptron

NPS Naval Postgraduate School

OSS open-source software

PIL Python Image Library

ResNet Residual Network

ROC receiver operating characteristic

SVM support vector machine

TFRecord TensorFlow Record

UAS unmanned autonomous system

VGG Visual Geometry Group

VGGNet VGG Network

Acknowledgments

While my name goes on this thesis as the author, I owe a number of others credit and gratitude for pouring time and effort into guiding my work directly or by carrying other burdens so that I could focus on thinking, coding, and writing. I will start with the latter. To my lovely bride, Adrienne, you have been the epitome of patience and grace as I poured countless hours into this work. Thank you for often single-handedly completing our house chores and understanding the long hours necessary to write well. Thank you for reading over my chapters for edits when I was too tired to read them! I am lucky to have you on my team.

Mathias, thanks for your approachability, guidance, thought-provoking critiques, and willingness to stick out the last year advising me from afar. You did well balancing new work, a young family, Google Hangouts thesis sessions, and lots of reading. Thank you for giving me the freedom to run with my vision for the thesis but providing thoughtful and necessary corrections along the way.

Michael, your edits added significant luster to my writing. Thanks for all of the reading and effort required to make such worthwhile corrections.

Thanks to the numerous Naval Postgraduate School professors (especially Dr. Marcus Stefanou, Dr. Neil Rowe, and Dr. Lyn Whitaker) who provided meaningful instruction and helped spark my interest in the field of machine learning.

And a final thanks to my office mates, Justin and Raven, who engaged in thoughtful conversations about life and neural networks, which made the time at school more enjoyable and helped grow my understanding of the material presented in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Today, thorough media analysis for the purpose of gathering intelligence proves time consuming and expensive. Human fascination with capturing and sharing experiences, the arm's reach availability of modern cameras, and the increasing amount of data posted to social media daily, feed a boundless flow of new images and videos. "Clara," a notional intelligence analyst, must wade through this flow of information searching for the critical imagery that will give her team the necessary tip to prevent the next homegrown terror attack. She sorts through her photographs and video stills diligently, fast enough to get through her minimum for the day, yet methodically enough to be sure to tag all relevant objects in an image for further analysis. Just one of many, Clara's efforts aggregate with thousands of others who stand watch over their nation hoping to find actionable intelligence in the day's social media postings. Without accurate and efficient computer algorithms to help, organizations within the U.S. Department of Defense (DOD) are left training human beings to complete repetitive and headache-inspiring image-tagging tasks. Fortunately, recent improvements in modern computer algorithms for complex image classification tasks elevate computer performance to a level comparable with humans; Russakovsky et al. [1] find that their best expert annotator outperforms modern algorithms by only 1.7%. For the DOD, this suggests computers can assist with, or take over, many of the organization's intelligence-related media analysis tasks today.

1.1 Current Image Classification Environment

To support the claim that computers stand ready to assist in these tasks, we explore recent developments in the image classification research environment in terms of the available algorithms, hardware, and software. For this thesis, image classification will refer to the automated process of labeling an image with single or multiple labels to describe what objects appear in that image. In the past decade, computer algorithms have gone from achieving noteworthy performance on simple image classification tasks to competing with human performance on complex tasks, such as labeling objects in an image into 1,000 possible categories for the ImageNet Large Scale Visual Recognition Challenge

(ILSVRC) [1]. Specifically, recent breakthroughs in the field of deep learning neural networks reduced the error rates of state-of-the-art systems by 30% to 50% [2] when completing tasks like those included in the ILSVRC. The most successful models harness deep convolutional neural networks (CNNs) for their architectures. They also take advantage of graphics processing units (GPUs) to speed up computations for training the networks by one or two orders of magnitude [2]. While the code and intricate algorithm architectures of such cutting-edge tools is beyond what an inexperienced programmer typically can achieve, the recent trend of industry and academia making machine learning software available via open-source licenses brings access to this technology to even novice programmers. Further, well-known companies like Google and Microsoft have already open sourced their machine-learning tools, TensorFlow and Microsoft Cognitive Toolkit (CNTK), respectively. Table 1.1 is a list of some of the major frameworks currently offered as open-source software (OSS). The progress in research and the availability of the source code for powerful frameworks should inspire reflection for large government organizations like the DOD. Is the DOD successfully developing similar technology to provide capabilities like autonomous image analysis and should the organization pursue available OSS solutions?

Table 1.1. Sample of open-source deep learning software frameworks

Organization	Framework	First Released
Online Community	Torch [3]	Oct 2002
Academic Community	scikit-learn [4]	Feb 2010
University of Montreal LISA Lab	Theano [5]	Mar 2010
Berkley Vision and Learning Center	Caffe [6]	Oct 2013
Google	TensorFlow [7]	Nov 2015
Academic Community	MXNet [8]	Dec 2015
Microsoft	CNTK [9]	Jan 2016

1.2 Using an Open-Source Deep Learning Software Library in the DOD

With the volume and variety of available machine learning algorithms that are continually improving, compounded with the challenges large government groups face in maintaining their software, organizations within the DOD seem unlikely to rapidly develop and employ cutting-edge machine learning systems through the Defense Acquisition System. While

the U.S. Government is certainly capable of undertaking complex software projects, they are often costly and software development continues to prove problematic [10]. Potentially limiting factors come to mind when considering government software acquisition. To name a few:

- Budget and contract constraints
- Complexity of the Defense Acquisition System
- Inability to modify source code behind a vendor produced system [11]
- Long contractual and acquisition timelines
- Limited machine learning talent pools

These limitations inspire the sentiment that software contracted for independent development by a DOD organization is more likely second rate, quickly obsolete, and too inflexible to take advantage of new breakthroughs in the field of deep learning.

On a positive note, current DOD policies towards OSS acknowledge the aforementioned limitations and encourage the consideration and application of open-source solutions whenever they are available. Specifically, the DOD Chief Information Officer (CIO) published a memorandum providing guidance to encourage market research of OSS solutions to meet mission needs in 2009 [11]. This and other policies facilitate the DOD employing state-of-the-art deep learning software through OSS frameworks. Given the current machine learning environment and a supportive stance by the DOD on OSS, we propose an evaluation of Google's TensorFlow as a viable solution to meet the DOD's machine learning needs for automated image classification tasks.

1.3 Why TensorFlow?

After a simple comparison of capabilities, we chose Google's TensorFlow to conduct an evaluation of a single framework through several experiments instead of comparing available frameworks across a single task. With numerous deep learning software frameworks to choose from and a desire to avoid benchmark testing, we select TensorFlow for our experiments and leave the evaluation of other frameworks for future work.

The decision to evaluate TensorFlow requires a brief discussion of the software's background and the factors supporting its selection. In November of 2015, Google released TensorFlow,

"an interface for expressing machine learning algorithms, and an implementation for executing such algorithms" [7] under the Apache 2.0 open-source license. Google developed its GoogLeNet neural network utilizing TensorFlow, winning the ILSVRC 2014 [7], the current image classification benchmark competition. The company also continued to make more accurate iterations of this trained network, codenamed Inception [12], [13], easily available for testing through the software framework. Above offering effective models, TensorFlow carries with it the financial support and technical talent of Google to keep the software improving. A confidence in Google's ability to push the limits of artificial intelligence through machine learning quickly inspired an online community, through GitHub [14], to partner with the company to improve TensorFlow. Figure 1.1 depicts the number of updates made to the software by Google employees and this online community over TensorFlow's first year as an open-source project. In summary, TensorFlow's implementation of cutting-edge machine learning algorithms, documented state-of-the-art performance in image classification tasks, and its swift adoption by the open-source community indicate that TensorFlow will stay relevant and state-of-the-art for the foreseeable future.

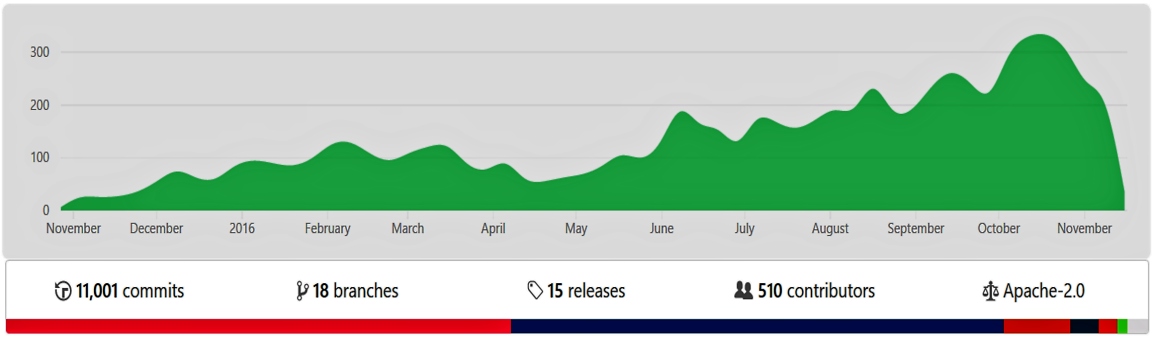


Figure 1.1. Number of commits made to the TensorFlow software library during its first year as an open-source library. A commit includes any instance of additions or subtractions to the code base accepted by the project's administrators. For TensorFlow, this includes addition of new models, interfaces, and tools for the software. TensorFlow had 15 versions released in its first year, marking several significant and rapid improvements [14]. Adapted from [14].

1.4 A Vision for TensorFlow-Based Intelligence-Related Media Analysis across the DOD

This section develops our vision for the applications of TensorFlow (or another OSS framework from Table 1.1) with respect to the challenges of modern intelligence-related media analysis in the DOD. Starting at the tactical level, a team of intelligence analysts at a battalion or squadron could benefit from a software solution like TensorFlow. Current methods require personnel to manually sort through thousands of images and hours of video to pull out applicable information from confiscated hard drives or collected intelligence, surveillance, and reconnaissance (ISR) [15]. What if analysts instead simply searched through images or stills from video, first tagged by a TensorFlow-based algorithm, that were specific to the threat they were looking for, like weapons that could indicate the militarization of an owner of a hard drive? Beyond a single unit, all four service branches would have something to gain from an intelligent unmanned autonomous system (UAS) that implements TensorFlow solutions. Most modern UASs simply broadcast ISR imagery back to a central location, consuming large amounts of resources in a constrained bandwidth environment in the process, and then require a human-in-the-loop for analysis. What if a surface-based UAS, far in front of a carrier strike group, passively conducted video surveillance and only broadcast imagery back across a satellite link when it determined that collected imagery contained a threat? Finally, recall the hypothetical analyst Clara, who works at a national level intelligence organization: a single analyst cannot possibly examine enough images from a day's typical posting to allow her agency to manage the flood of media created daily. Thus, the risk of missing warnings of terrorism, and threats against the citizens the agency supports, remains high. What if her department instead fed this flow of imagery through compute resources, efficiently implementing cutting-edge machine learning algorithms in TensorFlow, and pulled out images of interest as determined by current threat profiles. Analysts like Clara could then carefully examine these manageable streams of vetted images to pull out actionable information. This vision only touches upon a handful of use cases for TensorFlow in the DOD in a single problem domain. Even still, the potential benefit for successfully implementing the software in the organization sufficiently motivates our research to determine whether such use cases are feasible.

1.5 Problem Statement

As illustrated above, training humans to examine the intimidating number of images available for DOD intelligence analysis—quickly enough to ensure the information remains relevant—requires resources that many groups in the organization simply do not possess. These groups, from strategic-level national intelligence agencies to tactical-level military units, continually face time and manpower constraints in tackling media analysis challenges. In contrast to manpower, through the selection of cloud-based services offered commercially, the cost of compute resources continues to decline. Their affordability, coupled with the trend of commercial and academic researchers open-sourcing cutting-edge algorithms and software libraries, presents groups throughout the DOD with an opportunity: combine cheap compute resources with powerful state-of-the-art software to overcome time and personnel shortfalls. Before the DOD can exploit this opportunity, preliminary research is necessary to explore the merits of open-source machine learning software and algorithms when applied to specific DOD media analysis challenges. Through this thesis we set out to provide a starting point for future research and a baseline of support for the application of these open-source solutions in the Department of Defense.

1.6 Research Questions

To further describe the goals of our research and offer specifics of what we set out to accomplish, we present six research questions below. We will answer these questions through comparing the performance of algorithms implemented in TensorFlow to three previous Naval Postgraduate School (NPS) theses for identifying AK-47s [15], ships [16], and screenshots [17] in image databases.

1. Compared to previous methods, do TensorFlow-based deep CNNs: improve classification error rate; reduce training time and the size of datasets required for training; provide versatility and ease of use in different problem domains; and scale to enable deployment of trained networks for classification of images in resource-constrained environments?
2. Do TensorFlow's pretrained Inception models offer higher out-of-the-box recall and lower false positive rate (FPR) than previous methods to detect AK-47s and can they perform classification rapidly in a resource-constrained environment?

3. How does TensorFlow's pretrained Inception-v3 model perform in detecting ships in images when retrained with transfer learning methods compared to the error rate of previous methods?
4. How does a deep CNN, built and trained through TensorFlow with images acquired from an internet database, perform in identifying screenshots in an image in comparison to previous methods in terms of recall and FPR?
5. What are the potential benefits and risks of the DOD employing an open-source, continually updated, and public platform for its machine learning needs?
6. How are the results of this analysis relevant to the DOD and future-related research utilizing TensorFlow or other open-source frameworks?

1.7 The Way Ahead

Having introduced the problem and the motivation of our research, establishing the waypoints for answering our research questions seems helpful. In Chapter 2, we will discuss acquiring open-source software through the Defense Acquisition System, three previous NPS theses that will serve as our baseline for performance comparison, and modern algorithms and techniques we will implement in our research. Chapter 3 establishes the methodology for our three experiments and Chapter 4 provides their implementation details and the processes followed to conduct the experiments. With our tests complete, we present and discuss our results in Chapter 5. Finally, in Chapter 6 we draw conclusions from our results and specifically answer our research questions. The end state of this thesis is to present TensorFlow for employment by the DOD through introducing supporting policies, discussing the software's key terminology and capabilities, and exploring three potential use cases by comparing performance to previous methods. In accomplishing these three objectives, we hope to provide a logical and empirical argument for further application of open-source machine learning software in intelligence-related media analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter establishes the necessary background for our research by surveying four areas. First, Section 2.1 describes the Defense Acquisition System (DAS), along with software development and commercial software acquisition in this system, to garner support for applying OSS in the DOD. Next, Section 2.2 presents previously conducted experiments at NPS relating to computer-based object detection in images. These experiments serve as a starting point for performance comparison and potential applications of TensorFlow within the DOD. Section 2.3 provides a brief overview of CNNs and three modern algorithms that implement them in order to achieve state-of-the-art performance on complex classification problems. A final section explores some of the methods available to train these algorithms on new or organization specific image classification tasks. Through these sections, we demonstrate that an environment exists that encourages the acquisition of open-source solutions, like TensorFlow, to meet the DOD's needs for object detection in images.

2.1 Software Development and Commercial Software Acquisition in the DAS

The Department of Defense has a system in place to develop and deploy new products, including software, to its members. The system is known as the Defense Acquisition System or DAS. This section introduces the DAS as it would relate to the DOD's procurement and the contractor development of a proprietary equivalent to TensorFlow. We aim to show that developing such software through the system proves less efficient than simply incorporating an open-source offering. Understanding the challenges associated with developing cutting-edge software through the DAS requires an overview of the system and its past performance in completing complex projects. These challenges are well known and inspired the DOD to develop a preference for finding commercially available solutions to meet capability requirements. Moreover, recent defense-policy shifts that rebrand OSS as commercial software encourage pursuit of OSS products to meet the Department of Defense's needs. Finally, an analysis of the benefits and risks of incorporating OSS in defense systems gives confidence in the merit of considering such software before other options.

2.1.1 Overview of the DAS

To start, we present a definition of the DAS from its defining directive: "the management process by which the Department of Defense provides effective, affordable, and timely systems to the users" [18, p. 4]. The DAS is one of three components that make up the overarching system known as "Big A" acquisition [19, Ch. 1-3.2], which guides the incorporation of new products into the DOD. Since the other two components focus on top-level policy and budgeting, our remaining discussion will focus on the DAS, the process by which a software system progresses from need identification to delivery to DOD users [20]. Figure 2.1 provides the highest level overview of the DAS. It consists of a series of required documents that drive three milestone decisions for moving a system towards production and deployment. The current DAS is a product of numerous rules and regulations, added between 1947 and the present by congressional reform [21]. While the system provides structure and order to a challenging and fluid development process for complex weapon systems, the volume and variety of requirements can prove overwhelming. An overarching purpose behind the DAS, and its many reforms, is to minimize the loss of taxpayer dollars in developing DOD systems. The process achieves this goal through increased accountability and risk mitigation. These characteristics, coupled with over-regulation, stifle rapid innovation. Although well intentioned, and even necessary, the system of complex regulations that make up the DAS becomes an added source of friction in an already complex product-development process.

2.1.2 Frustrations in Software Development through the DAS

The attributes described in Section 2.1.1 contribute to a negative sentiment towards pursuing software intensive projects through the DAS. Speaking of the numerous studies conducted to identify problems and suggest reform to acquisition processes, Linda Levine and Bill Novak of the Software Engineering Institute at Carnegie Mellon University summarize these frustrations well: "It is disturbing, however, that many problems associated with the development and acquisition of software-intensive systems remain unresolved–and growing in magnitude–while proposed solutions remain either untried or not sustained" [10, p. 1]. Others suggest that the system, conceptualized and established during World War II, is simply too outdated for the demands of 21st Century development projects, regardless of reforms [22]. Such sentiments have led to policy shifts that encourage acquisition of commercially produced systems, including commercial software, before development of

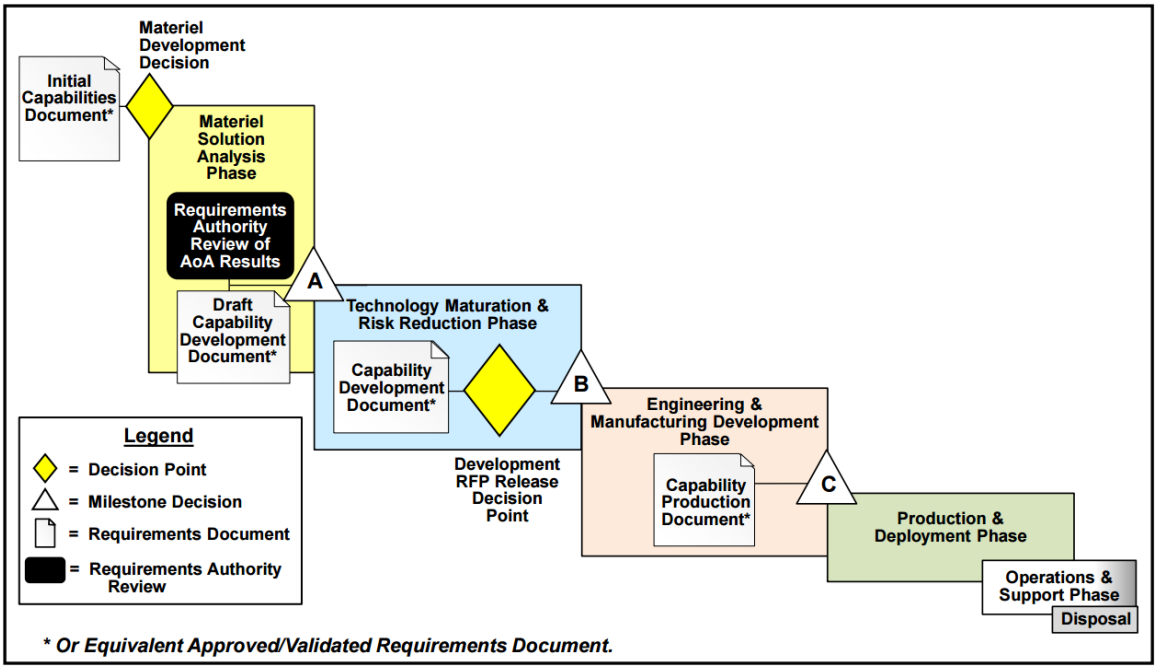


Figure 2.1. Overview of the Defense Acquisition System process. Source: [20].

new systems through the DAS. For example, the latest version of Department of Defense Directive (DODD) 5000.01, which "provides management principles and mandatory policies and procedures for managing all acquisition programs" [18], explicitly states that DOD Components shall consider commercially available products prior to other acquisition options [18]. With supportive policy in place to acquire commercial software, the determination of whether or not such policies apply to OSS remains.

2.1.3 Using OSS in the DOD

This section shows that the DOD considers OSS a commercial product for purposes of acquisition and also discusses the benefits and risks of implementing OSS in defense systems. For the average DOD user or program manager, deciphering the differences among the types of software available (i.e., commercial, freeware, shareware, open-source, closed-source) and what variants they can use with government systems proves perplexing. Fortunately, policy exists that clears up this confusion for the defense professional hoping to incorporate an OSS solution into an acquisition project. In 2009 the CIO of the DOD characterized OSS as equivalent to commercial computer software and directed that defense

programs include it in the mandated market research of commercially available solutions discussed in Section 2.1.2 [11, p. 4, par 2.b]. This policy grants permission for the acquisition professional to implement OSS in defense programs using the same regulations that govern closed-source software.

With permission established, we will now weigh some benefits and risks of OSS. Specifically, we examine the software support available and cost, the speed of updates and improvements, and the security and long-term use of such software. Open-source is not synonymous with free. Often successful OSS projects, like Linux Red Hat, offer support contracts that enterprise level users can purchase, even though the source code for the software is available publicly. A significant risk for the DOD implementing OSS is that many projects do not provide this purchasable support package. While a responsive online community, thorough documentation, and hirable third party support can remedy this risk, the level of support may fall short of the what companies like Microsoft offer to the DOD for closed-source software. In cases that deem a maximum level of support essential, it is possible to hire a third-party who is familiar with the OSS code base and applications. The risk associated with reduced support is acceptable in many cases due to the prospect for cost savings. As current DOD policy states: "Since OSS typically does not have a per-seat licensing cost, it can provide a cost advantage in situations where many copies of the software may be required, and can mitigate risk of cost growth due to licensing in situations where the total number of users may not be known in advance" [11, p. 4]. Also, once the DOD establishes guidance for employing an OSS project, other components of the organization can quickly adopt the software, since the acquisition of per-instance licenses is no longer a concern. Despite support concerns, the cost savings and speed at which organizations across the DOD can share software solutions makes OSS an attractive alternative to other forms of commercial software.

The DOD desires innovation but struggles to acquire innovative systems. OSS can help with faster update cycles and the ability to tailor source code for specific needs and mission types. As illustrated by the fifteen releases of TensorFlow in its first year, see Figure 1.1, popular OSS projects attain release cycles measurable in terms of weeks or months compared to the annual or bi-annual releases of other types of commercial software. The short update cycle of OSS, coupled with the fact that new versions do not require purchase of updated licenses, affords the defense sector an opportunity for rapid and cost efficient prototyping

of new software solutions [11]. Further, the varying missions and threats faced by DOD Components demand software that users or support personnel can modify swiftly and apply to new problems or adapting enemies. The DOD CIO summarized this benefit well in his 2009 memorandum: "The unrestricted ability to modify software source code enables the Department to respond more rapidly to changing situations, missions, and future threats" [11, p. 4]. But what are the risks? With a community of users producing rapid updates, the potential for the introduction of code breaking bugs exists. Fortunately, modern software testing techniques and issue reporting services offered through websites hosting open-source projects, like GitHub [14], largely alleviate this risk. These techniques, the number of reviewers and testers of popular projects, and the ability to delay implementing the new releases of software for critical defense systems, allow an organization to manage the unique uncertainties of relying on OSS. With benefits of faster updates and the ability to directly modify source code outweighing other concerns, OSS provides the DOD with an opportunity to maintain state-of-the-art software solutions across its many mission sets.

We must also weigh benefits and risks in terms of security and the prospect for long-term use of OSS products. One unique concern of OSS is that an adversary, hoping to find a vulnerability to attack a defense system, has access to the source code that drives the system. While the dangers of such access seem obvious, perhaps it is most surprising that some experts argue that OSS leads to improved security over closed-source software. This argument's merit stems from the fact that benevolent users, such as DOD security professionals, also have access to the source code, instead of only the private firms developing the software. This access promotes efficient discovery of security flaws, which an open-source community or a DOD organization can quickly patch and release. In contrast, the license agreement of closed-source software would make such modification and distribution illegal [23]. Moving on to long-term use, some opponents may argue that a group maintaining an open-source project can stop updating the source code at anytime, thus introducing risk in relying on such software. To counter this argument, first consider the repercussions of a vendor owning legal rights to the source code of a defense system going out of business. Next, think of a community dropping support for an open-source project, of which the DOD maintains legal access to any copy of the source code it possesses [23]. In the former scenario, the DOD holds few options beyond running the current version of the software until paying a new vendor to develop a comparable product from scratch. In

the latter circumstance, DOD users can assess their need for further updates of the OSS before deciding whether or not to internally maintain a version of the software or contract out a vendor to pick-up where the open-source community left off. This right to update and modify the source code frees the DOD from vendor lock and provides greater opportunity for long-term use and improvement of OSS products.

With updated policy in the DAS and from DOD leadership in support of the pursuit of OSS, defense professionals have the necessary permission to implement OSS solutions to meet their operational needs. The frustrations of developing complex software through the DAS provides the motivation to seek OSS solutions prior to software development with the system. Finally, an analysis of top-level benefits and risks of engaging OSS in defense suggests its suitability for many defense systems. Overall, this assessment inspires confidence in the DOD's ability to acquire, implement, maintain, and benefit from an OSS library like TensorFlow.

2.2 Previous NPS Image Classification Experiments

In addition to weighing the benefits of OSS and presenting the ability of the DOD to employ a solution like TensorFlow, we aspire to display empirical evidence that the software performs in realistic DOD use cases. Following the AK-47, ship, and screenshot detector scenarios developed in Section 1.4, we introduce three previous experiments at NPS which evaluate machine learning algorithms on datasets related to our scenarios. We eventually incorporate these datasets into our experiments and employ the results from the prior tests to serve as a baseline for comparing TensorFlow's performance on the same image sets. These three experiments all rely on human-engineered feature extractors specific to each object detection problem. While their three algorithms contain some similarities, the methods are mostly unique to each experiment. The discussion of this varying methodology in this section stands in contrast to the single algorithm type, a deep CNN, and a single software library that we apply across all three datasets for our experiments. Finally, we also include any training or evaluation time metrics reported in the previous theses to compare to our tests.

2.2.1 Detection of AK-47s by Justin Jones

The first experiment we reference simulates the potential use case for an intelligence analyst at a battalion or squadron level discussed in Section 1.4. Justin Jones [15] published the results of this experiment at NPS in September of 2010. Starting with 18 videos containing AK-47s, Jones [15] developed a training set and test set of images by capturing stills from these videos. His training set consisted of 1,146 stills containing AK-47s from 13 of the videos and 5,660 negative training images from the internet. His test set contains 687 stills of AK-47s from five videos and 7,045 images containing people and other objects without AK-47s present. To conduct his experiment, Jones [15] implemented both whole AK-47 and left and right parts-based classifiers with three algorithms. First, he trained several iterations of a Viola-Jones Classifier to pull out features from small portions of the images with the Haar training utility from OpenCV [24], an open-source computer vision software library. He then fed the extracted features from an image segment into a support vector machine (SVM) and a simple artificial neural network, known as a multilayer perceptron (MLP), to determine if an AK-47 existed in the segment. Repeating this process for over 300,000 unique windows across an image, he produced an AK-47 count for each video still. Table 2.1 contains training resources available and time required for each algorithm Jones [15] tested.

Table 2.1. Training time statistics from Jones' [15] thesis.

Algorithm	**Processor**	**RAM**	**Training Time**
Viola-Jones Classifiers	Intel Core 2, 2.4 GHz	2GB	48 hours
SVM	Intel Core 2, 2.0 GHz	4GB	2 seconds
MLP	Intel Core 2, 2.0 GHz	4GB	2 seconds

Jones [15] provided the times in this table for training the components of his algorithms on his 6,806 total positive and negative training images.

Jones [15] measured the performance of his algorithms through a comparison of recall and FPR on the test image set described above. He presented this information in the form of a receiver operating characteristic (ROC) curve and Figure 2.2 displays his overall results for his algorithms. This curve will serve as the basis for our performance comparison.

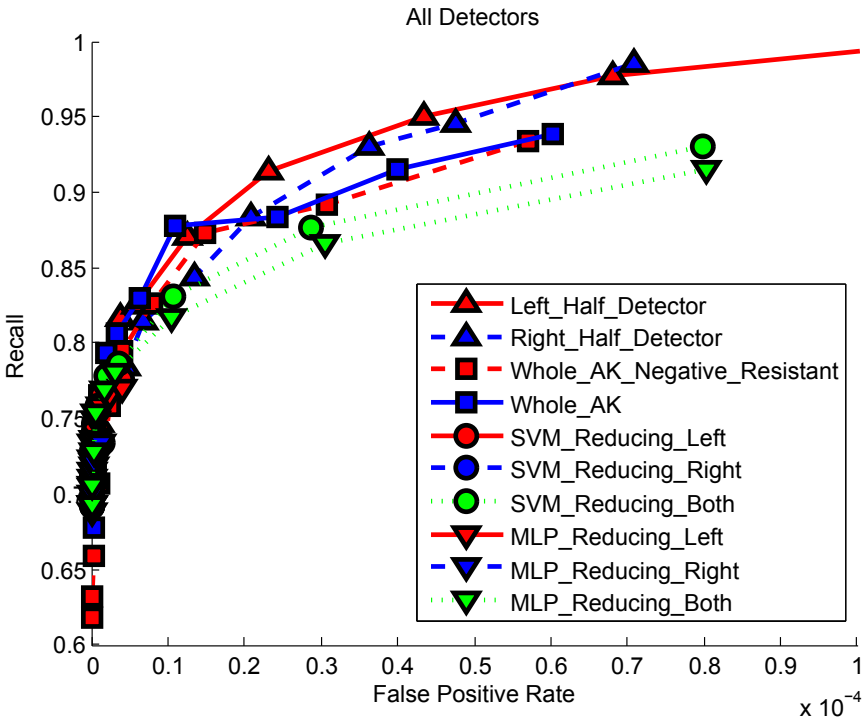


Figure 2.2. Jones' [15] AK-47 detector performance. Notably, Jones' [15] scale for his FPR is at 10^{-4}. His algorithm produced a classification at over 300,000 window positions across each frame to predict how many AK-47s existed in an image. The importance of limiting false positives with such a large number of classifications per image led Jones' [15] to only display results with an extremely low FPR. Source: [15, Figure 5.8].

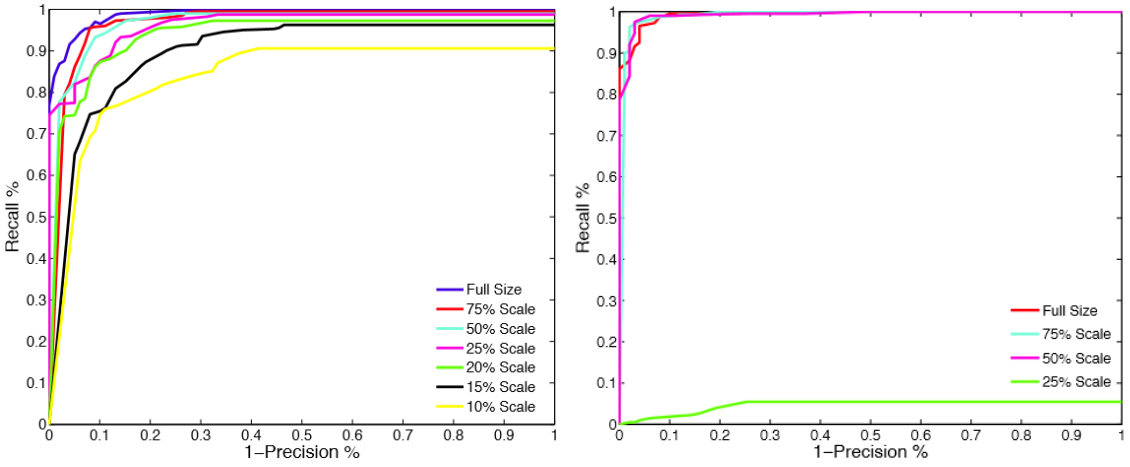
2.2.2 Detection of Ships by David Camp

The second experiment we reference simulates the use case for intelligent UASs discussed in Section 1.4. David Camp published the results of this experiment at NPS in December of 2013 [16]. Camp's archived training image set from this experiment consists of 110 instances of ships and 97 instances of sky, sea, and coastline with and without buildings that do not contain ships. These numbers vary slightly from his reported 105 images for both categories. His test set possesses 405 images containing ships and 100 images not containing ships, similar to the training set. Camp scaled all test images at eight levels, from full scale to five percent scale, to simulate classification at different distances from a sensor. He applied several feature extractors and methods for classification to determine which provided the

strongest performance, but for the sake of brevity we only discuss his top two classifiers. The first is what Camp labeled a HYBRID detector, combining histogram of oriented gradients (HOG) and bag-of-words (BOW) algorithms into one classifier. The second was a deformable parts model (DPM). Like Jones, Camp also employed the OpenCV software library for all of his experiments. To measure performance, Camp calculated recall and FPR to produce a ROC curve. Table 2.2 presents information about runtime performance of the algorithms while Figure 2.3 presents his ROC curves for comparison of classification performance.

Table 2.2. Evaluation runtime statistics from Camp's [16] thesis for a single image.

Algorithm	**GPU (RAM)**	**Evaluation Time**
HYBRID	NVIDIA Quadro 2000 (1GB)	0.5 seconds
DPM		2 seconds



(a) HYBRID ROC curve

(b) DPM ROC curve

Figure 2.3. Camp's [16] best ROC curves at varying scales for performance comparison. We include both ROC curves as the HYBRID algorithm performs best across all scales, but the DPM algorithm achieves the best performance at larger scales. While Camp labeled the x-axis with $1 - Precision$, from his paper he describes this value as equivalent to the FPR. Both algorithms show a significant decrease in performance on smaller scales. Source: [16, Figure 55, Figure 18].

2.2.3 Detection of Screenshots by Lauren Sharpe

The final experiment for comparison autonomously detects whether or not an image is a screenshot. This experiment simulates the use case for a national level agency that must sort through volumes of images to identify those that meet a specific threat profile or criteria. In this case, the images sought may be from a terrorist group hiding instructions for its members in posts of a screenshot. A group might do this versus posting via traditional methods if they believe the U.S. Government is searching text posts autonomously. Separately, if this agency was scanning collected hard drives, screenshots could require human inspection to determine if a drive's owner was producing manuals or instructions sets for nefarious purposes, like bomb making. In contrast to marking a screenshot for review, this tool could also serve to filter out screenshots from further analysis by humans if an agency considered them noise. Lauren Sharpe [17] first published the results in this section in her NPS thesis in June of 2013. For her work, Sharpe [17] collected screenshot images from the Wikimedia Commons' user-tagged *Screenshots* category [25]. This source produced a positive image set of 2,423 images. For negative examples, Sharpe [17] utilized the *13 Natural Scene Categories* dataset [26], containing scenes of furniture in homes, highways, buildings, coastlines and more. The total non-screenshot examples, or *other* category, amounted to 3,694 images. Her negative example set only contains grayscale images while her screenshot examples contain color images. Sharpe [17] deployed OpenCV, the Python Image Library (PIL) [27], and Orange [28] software libraries to implement her algorithms. She extracted multiple feature sets with OpenCV and PIL and fed them into an a Naïve Bayes Classifier from the Orange library. She implemented ten-fold cross-validation across 2,400 positive and 2,400 negative example images to evaluate her algorithm. Table 2.3 depicts both the training and evaluation runtime statistics of Sharpe's [17] best performing algorithm. She combines these times due to her cross-validation method. The four feature sets she manually selected and tested are: Line Segment Percentages (1), Line Segments Binned by Length (2), Intensity Entropy (3), and Intensity-Based Histograms (4). Table 2.4 depicts the performance of Sharpe's [17] algorithm testing 11 combinations of these feature sets. Also, Figure 2.4 depicts a ROC curve to display performance of the first feature set only. These are the metrics that will serve as a baseline for comparing performance to our algorithm implemented in TensorFlow.

Table 2.3. Training and evaluation runtime statistics from Sharpe's [17] thesis.

Algorithm	Images	VM Resources	Train & Test Time
All Features	4800	32-bit VM with 1GB RAM	14 minutes
All Features	1	32-bit VM with 1GB RAM	0.18 seconds

Sharpe [17] ran her algorithms in a virtual machine running on a Windows 7 system with 8GB RAM and an Intel Core i7-2600 processor. She did not provide further details than those listed in this table as to the total resources this virtual machine had access to from its hosted system. (i.e., number of processor cores)

Table 2.4. Sharpe's [17] screenshot detector results for combinations of four feature sets. Source: [17, Table 4.9].

Set 1	Set 2	Set 3	Set 4	Line Bins	Color Bins	Accuracy	Precision	Recall	F-score
–	–	Yes	Yes	–	50	0.942	0.950	0.933	0.941
–	Yes	–	Yes	10	500	0.968	0.993	0.942	0.967
–	Yes	Yes	–	10	–	0.937	0.994	0.880	0.933
–	Yes	Yes	Yes	10	500	0.966	0.991	0.941	0.965
Yes	–	–	Yes	–	500	0.963	0.989	0.937	0.962
Yes	–	Yes	–	–	–	0.954	0.963	0.945	0.954
Yes	–	Yes	Yes	–	10	0.967	0.981	0.953	0.967
Yes	Yes	–	–	10	–	0.956	0.973	0.937	0.955
Yes	Yes	–	Yes	50	500	0.975	0.997	0.952	0.974
Yes	Yes	Yes	–	10	–	0.973	0.987	0.957	0.972
Yes	Yes	Yes	Yes	10	10	0.980	0.997	0.963	0.980

Sharpe [17] conducted experiments for the 11 combinations of her four feature sets in this table. For each combination containing the feature set Line Segments Binned by Length (2) or Intensity-Based Histograms (4) she varied the number of *Line Bins* and *Color Bins*, respectively. She tested five bin sizes (10, 50, 100, 500, and 1000) for each. The line bins grouped line segments by their length and the color bins grouped pixels in bins by pixel-value intensity. Sharpe [17] depicts only the best performing number of bins for each feature set combination in the table.

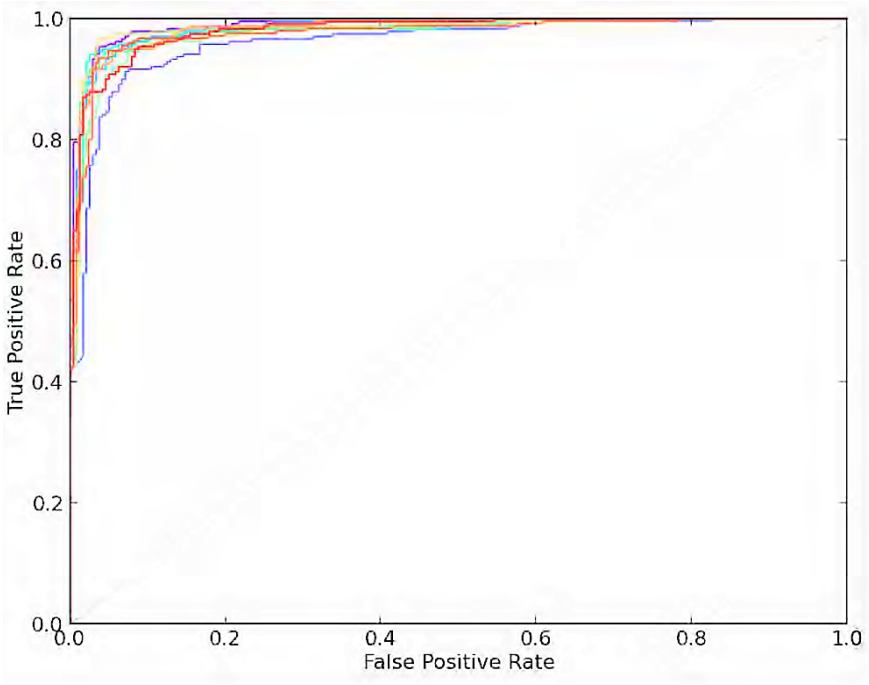


Figure 2.4. ROC curve for Sharpe's [17] first feature set. This is the only ROC curve that Sharpe [17] provided in her report. Even though it is not for the best performing algorithm, which combined all four feature sets, we include it for comparison to our ROC curves. Each line in the plot represents results for one of ten iterations of the ten-fold cross-validation Sharpe [17] conducted for each algorithm. Source: [17, Figure 4.16].

2.3 Overview of CNNs and Modern Models

This section describes the key ideas that make our experiments possible and the underlying technology behind the models we apply with TensorFlow. First, we present a swift introduction to the convolution operation, the convolutional layer, and the pooling layer as the building blocks for the CNNs from our experiments. The next section explores the concept of deep learning and its benefits when combined with convolutions in deep convolutional neural networks. A third section covers the importance of quality training data and some of the large modern datasets available. Fourth, Section 2.3.4 introduces and describes the specific models, already configured and available in TensorFlow, that we evaluate in the experiments detailed in Chapter 4. Finally, we discuss transfer learning and some of the key techniques available to improve performance of these models during training. A note

to the reader: we will interchange the common terms network, model, and architecture to describe the algorithms. These sections serve as the academic foundation for our research, which for the most part TensorFlow's application program interface (API) abstracts away, but these concepts remain essential to understanding what the software does.

2.3.1 Convolutional Neural Networks

The academic community has thoroughly documented artificial neural networks (ANNs) and their usefulness across a spectrum of data science problems and we will forgo an introduction in this paper. Instead, we focus on a specific type of ANN, known as a deep convolutional neural network, which sit at the core of most of the recent breakthroughs in computer vision related tasks [12]. The academic competition inspired by challenges like ImageNet, introduced in Chapter 1 and discussed later, fueled improved versions of CNNs [13], [29]–[31] with near human-level performance on complex image classification tasks [1]. Before we dive into details of these algorithms, we discuss the fundamental building block that makes a neural network convolutional; the convolution itself. We then give, a brief summary of other CNN components before moving on to a discussion of more complex modern CNNs.

The simple example in Figure 2.5 provides an intuition for the inner workings of a convolutional layer. Consider a vision algorithm designed to provide an autonomous system the ability to "see" a printed tic-tac-toe board. This ability will serve as a first step in building a complete system to compete with a human being. In our example image from the figure, each block in the image represents a pixel with a value of zero for white and one for black. Our algorithm will engage two convolutional *kernels*, or filters, that contain the patterns to match in portions of our image. By matching, we hope to identify key locations in the tic-tac-toe board. Through the convolution operation, which mathematically executes a dot product, a single output value gets produced for each window location in the image. Logically, we determine the spacing of the evaluation locations by starting in the upper left hand corner and repeatedly sliding the center of the window right by two pixels, a *stride* of two. We move the *corner-detection filter* (highlighted in blue) across the top three rows until reaching the right edge of the image and then start over again on the left side but two rows lower. This process continues until reaching the bottom right edge of the image. The *space-detection filter* at the bottom of the image (highlighted in red) mirrors the *corner-detection*

filter for this illustration. Typically, convolutional layers calculate the outputs for all filters across an entire image simultaneously. They do so through efficient matrix multiplication and addition. The gold boxes in the second row of the figure show the calculations that take place at each location to determine the output. We omit multiplications resulting in zero because of an image or filter value of zero and thus only the pixels in the yellow boxes affect the output of the convolution at this window position. The final row of the figure depicts the locations where our two filters produce their maximal output. These locations correspond to the corners defining the board for the *corner-detection filter* and the possible play locations for the *space-detection filter*. Figure 2.5 presents only the basic principles of the convolution for a two dimension, one channel example. An inquisitive reader can find an explanation of concepts like padding, convolution of multiple channel filters across the red, green, and blue channels of an image, and weight initialization in resources such as Stanford's online course on CNNs [32].

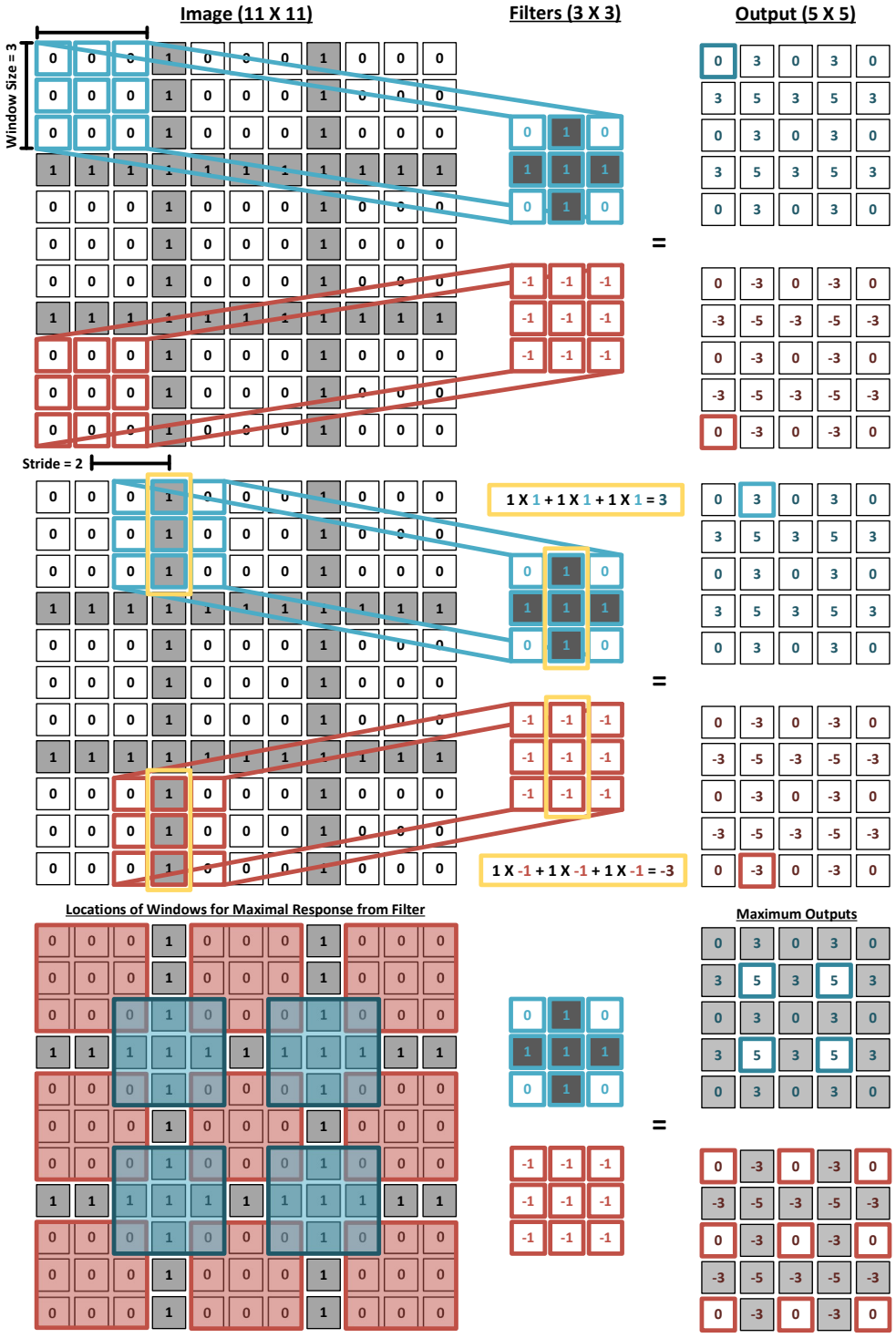


Figure 2.5. Simple example to introduce the convolution.

A few key concepts remain to complete an overview of CNNs (see Nielsen's online textbook for a more thorough discussion [33]). First, the filter values, also known as *weights*, are one component a CNN updates during training. These networks essentially learn, by labeled example images, to recognize patterns from raw pixel values that indicate a certain type of object exists in the image. The initial convolutional layers possess multiple filters with learned weights that draw out meaningful features from groups of pixels across an image; edges, color patterns, corners and more. Modern CNNs learn hundreds of different filters across their many layers, the first filters applied to the raw pixel input and later filters applied to the output values of previous convolutional layers. Each convolutional-layer output summarizes how strongly a learned pattern exists in a group of pixels or previous layer outputs. In our example from Figure 2.5, a single output value for each filter represents nine pixels at each evaluation location. The outputs of the convolution are run through an activation function which makes a determination if a pattern is prevalent enough to pass to the next layers. Second, the *pooling layer* is another key layer in CNNs [32]. These layers follow convolutional layers and reduce the output values passed to later layers. *Max-pooling* is a common pooling layer function and it simply takes a max of adjacent outputs from the convolutional layer, perhaps reducing four adjacent outputs to the largest output of the four. Convolutional and pooling layers then get repeated with several other innovative techniques, such as dropout [34], employed at certain layers. In this manner a CNN pushes only the most strongly detected feature information to later layers [2] to drive the algorithm towards a set of output neurons, each corresponding to a specific classification. Now that we have established a description of the convolution and the general construction of a CNN, we will proceed to an examination of what makes such networks "deep" and how this depth provides additional expressive capacity.

2.3.2 Deep Learning

We start with a definition from an expert in the field of deep learning, Yoshua Bengio: "A deep learning algorithm is a particular kind of representation learning procedure that discovers multiple levels of representation, with higher-level features representing more abstract aspects of the data" [2]. By simply repeating combinations of the layers discussed in the previous section, a CNN becomes deep. Compared to previous machine learning algorithms, deep learning allows an automated discovery of higher-level features and abstractions from raw input data. Deep CNNs do not require manual extraction of features

prior to feeding inputs into a neural network, as seen in the previous methods discussed in Section 2.2. These algorithms first learn low-level features, which typically look like simple edge and pattern detectors tailored to the training data [35]. These first features, closer to individual pixels than humans can inspect or identify efficiently, allow a neural network to then learn larger abstractions through combinations of the features. Referring back to Figure 2.5 and our tic-tac-toe example, a next convolutional layer may combine the abstractions of a playable space identified by the *space-detection filter*, surrounded at all four corners by intersecting lines identified from the *corner-detection filter*, as a new abstraction of the center playable space of the tic-tac-toe board. Expanding this simple concept of combination across many more layers, certain types of deep learning neural networks become surprisingly efficient at classification requiring a high level of abstraction [2]. For this reason, deep learning algorithms serve as the base for the significant improvements of artificial intelligence applications in the field of computer vision occurring rapidly in only the past few years [2].

With a brief introduction behind us, we will explore the unique benefits and challenges of employing deep architectures. The benefit, as cited in Chapter 1, is the reduction of state-of-the-art error rates by 30% to 50% on complex benchmark competitions [2], like ImageNet. Through this competition, the importance of network depth emerges as all recent top performing models possessed depths between sixteen to thirty layers [31]. The challenge lies in the fact that as networks get deeper they are more difficult to train [31]. More layers mandate more trainable parameters. As a result, deep CNNs depend upon thousands of labeled examples to train to convergence without overfitting [29]. Along with a need for more data, deep networks take longer to train and require specialized hardware to do so efficiently. Specifically, a programmer can now train deep learning algorithms across multiple GPUs, instead of a single GPU or many traditional CPUs, to speed up training by several orders of magnitude [2]. These hardware advances enable researchers to train hundreds of iterations of their algorithms, adjusting key hyperparameters like learning rate, to achieve optimal performance and reap the benefits of higher levels of abstraction [29]. Still, the complexity of modern architectures can lead to training times measured in days to weeks [30] to obtain the performance of the algorithms winning classification competitions. In summary, while the performance benefits impress, training deep algorithms proves difficult and limitations exist to simply continuing to increase the depth to further improve performance [31].

2.3.3 The Importance of Data and Publicly Available Image Sets

With increases in the complexity of deep CNN architectures, the importance of quantity and quality of training examples for learning also increases. Some experts even go as far as making the argument: "As a rule of thumb, a dumb algorithm with lots and lots of data beats a clever one with modest amounts of it" [36]. This section introduces techniques for augmenting smaller datasets and some publicly available image sets to serve as a starting point for training and testing algorithms. Several methods exist that reduce the number of ground truth examples necessary for convergence: smarter weight initialization; training first with artificial data [37]; and transfer learning [38], a method for repurposing already trained networks. Of these methods we specifically benefit from transfer learning, which we will examine in Section 2.3.4. Still, there remains a need for a large number of quality training examples when implementing deep CNNs. There must be enough labeled data to first train reliable low-level feature extractors and also build the mid-to-high levels of abstractions necessary to benefit from deep architectures. Furthermore, it can be difficult [39], but not impossible [35], to know what features a trained neural network is learning in order to classify an image. With too few examples, all training images could possess a similar trait (like the same background), unrelated to the object for detection, that an algorithm learns as an indicator of the object. Thus, if we want to be sure that the network learns the object we intend, our datasets must possess sufficient quality examples: that is, enough instances of an object in a variety of realistic settings. We must show an algorithm examples of that object in different lighting conditions, from different angles, and in the type of scenes in which we want to be sure our algorithm works. Fortunately, there are large and diverse datasets available to help overcome this challenge. Table 2.5 presents some of the largest available to aid the DOD in machine learning and image classification related research; our research specifically benefited from ImageNet.

A final dataset-related concept is the trend of complex networks to perform image preprocessing during training to artificially increase the size of the datasets in hopes of reducing overfitting [29]. Since the networks usually require a constant input size, some combination of scaling and cropping is necessary to make images fit in each network due to the variety of sizes and resolutions present throughout datasets [29]. Further, to ensure brightness invariance, networks may normalize pixel brightness across a batch of images or randomly adjust it to ensure that illumination of an object in an image does not affect its classification.

Table 2.5. A sample of online image sets for training deep networks.

dataset	# Categories	Images	DOD Related Labels
ImageNet [1]	1,000	1,331,167	*rifle, assault rifle, tank, bulletproof vest, and more*
COCO [40]	80	300,000	*airplane, car, truck, person*
Open Images [41]	6,000	9 Million	*warship, tarmac, rifle, and more*

This table lists some of the labels from each set that apply to the DOD. ImageNet is the most recognized and organized of the datasets, however it does not provide licenses for republishing the images. The Microsoft sponsored COCO is the most unique. It offers five captions for each of its images that attempt to explain how objects in the image are interacting. Open Images, created by Google, is a library of URLs to labeled image data that strives to contain only creative commons license images, lifting restriction on republishing and reusing the images.

Other techniques involving random distortions of the training images, like cropping, flipping from left to right, or scaling, can artificially increase the number of training examples by creating multiple unique observations from a single image [42]. Preprocessing methods vary by model and TensorFlow implements those prescribed by the model's designers while allowing a user to adjust the preprocessing steps as desired.

2.3.4 Modern Models and Performance Improvements

The trend of researchers to open source their image classification algorithms [13], [29]–[31] makes them available for others to apply to new problems with little effort. Even more convenient, a number of software libraries quickly implement the newest algorithms. This section introduces three successful deep CNNs, provides a brief summary of their key contributions, and performs a high level comparison of the algorithms. Experts apply many innovative techniques across the field of deep CNNs that the creators of these algorithms cover in cited papers but we omit from our discussion for the sake of brevity.

The first algorithm, trained by Oxford's Visual Geometry Group (VGG) and thus known as VGG Network (VGGNet) [30], secured a first and second place position in the ILSVRC 2014. There are three different versions of this network available for testing in many open-source libraries, varying the number of convolutional layers to provide network depths of 11, 16, and 19 while keeping other parameters the same. VGGNet's contribution came by showing that a network with very small convolution filters, three by three pixels as compared

to other networks with larger five by five or seven by seven filters [42], could achieve state of the art performance by simply increasing network depth. This depth increase was possible in the network due to the decreased number of trainable parameters required by the smaller convolutions [30]. Figure 2.6 depicts the largest VGGNet architecture in a side-by-side comparison with Microsoft's ResNet model and Table 2.6 presents a high level comparison of the network to another recent model.

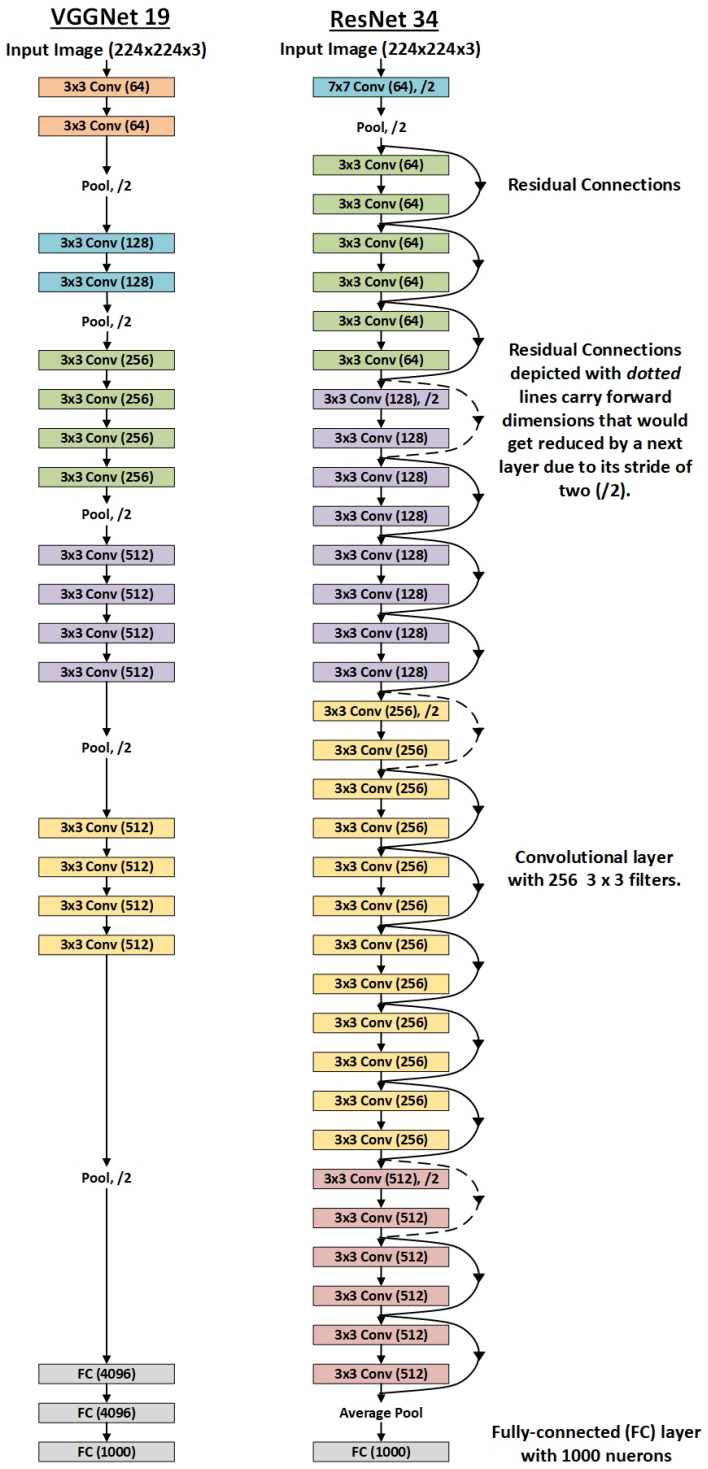


Figure 2.6. VGGNet and ResNet architectures. We recreate this figure from He et al. [31, Figure 3].

Microsoft's ResNet [31], which took first place in the 2015 competitions for both ILSVRC and COCO, is a second algorithm of interest. ResNet introduced the latest significant improvement for state-of-the-art CNNs [32] through its employment of residual layers. In seeking to develop deeper networks, and overcome some of the challenges mentioned in Section 2.3.2, Microsoft's research team sought to simplify the training process. The team created *residual layers* which they discuss in detail in their paper [31]. Here, we will only attempt to establish an intuition of the benefit from the residual layers. These layers simply forward the inputs of previous layers ahead two layers, as depicted by the arrows on the right of Figure 2.6, along with new information pulled out from the previous convolutional layers. In this manner a convolutional layer has all of the information provided to previous layers available to it and only has to extract additional information, or a residual mapping, if it improves classification performance. This affords a network architect the ability to increase the number of layers and know that a network will converge on the optimal number. Residual layers ensure this by allowing a layer to forgo extracting additional information and to simply pass the output of previous layers forward [31]. This method introduces efficiencies that lead to significantly shorter training times [13], [31]. Table 2.6 includes details of two ResNet models for comparison.

Google's Inception networks are the final model we introduce. The first version of the network, also known as GoogLeNet, won one of the ILSVRC 2014 challenges [42]. Since this competition, Google released several iterations of the network; we will focus on two. First, Inception-v3 [12], for its computational efficiency, making it suitable for use in resource-constrained environments, coupled with its state-of-the-art performance. Next, Google's latest iteration of the network, Inception-ResNet-v2 [13], which incorporates Microsoft's residual layers to further reduce classification error rates. Figs. 2.7 and 2.8 depict the architectures for these networks and Table 2.6 provides their key performance specifications.

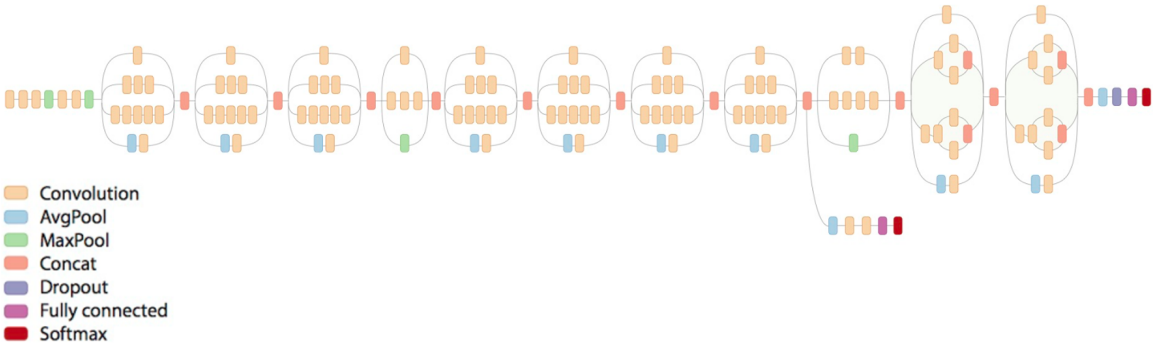


Figure 2.7. Illustration of the Inception-v3 architecture. From the left side of the figure, the Inception-v3 combines convolutions and max-pooling layers to quickly pull out relevant patterns from the pixel values to reduce the 299x299x3 input images to a 35x35x192 output after the second max-pooling layer. This smaller output represents the low-level features pulled from the image. The Inception-v3 model then feeds the smaller output through multiple paths of what Szegedy et al. [12] call *Inception modules*, to detect higher level abstractions from the features. The salmon-colored concatenation nodes in the diagram represent a stacking of the outputs from all of the Inception module paths to provide them to the next Inception module. Repeating several variants of these modules, finally the Inception-v3 model feeds the output to a fully-connected layer (second node from the right) with 1,001 neurons representing the 1,000 ImageNet categories along with a background category. Source: [43].

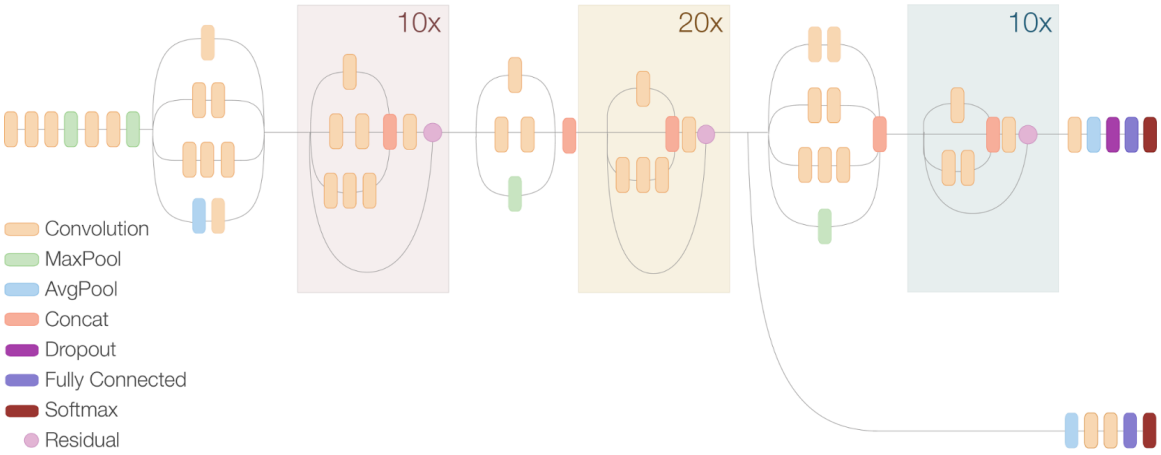


Figure 2.8. Illustration of the Inception-ResNet-v2 architecture. The Inception-ResNet-v2 model contains a similar architecture to the Inception-v3 model discussed in Figure 2.7. Two key differences are the fewer convolutions in each Inception module (defined in Figure 2.7) but significantly more repetitions of these modules to allow a greater network depth. This figure depicts the residual layer connections (introduced by He et al. [31]) as purple circles after three variants of Inception modules. These residual connections facilitate the blocks of 10 or 20 repetitions (depicted in the color boxes) for the three modules. The first six and last five nodes of the Inception-ResNet-v2 architecture match Inception-v3. Source: [43].

The models presented in this section are complex, but their success at solving challenging image classification tasks makes them worth studying. Teams of experts in the field of machine learning developed each of these algorithms. Few individuals, or even teams, could independently create and employ the architectures and methods necessary to achieve similar cutting-edge performance. Fortunately, since these architectures are made available publicly by their creators, and then implemented in software through open-source libraries, groups that could not produce them independently have the ability to employ them.

2.3.5 Training Methods

With a discussion of state-of-the-art CNNs's performance on the ImageNet dataset completed, in this section we will look to applying them to organization-specific image classification tasks. As discussed earlier, these networks require large amounts of quality training data to be successful. If an organization has enough labeled training data available, it

Table 2.6. A top-level overview of three deep CNNs.

Algorithm	**Top-1 Err. (%)**	**Top-5 Err. (%)**	**Parameters**	**FLOPs**
VGGNet 19	24.7	7.1	144M	19.6G [31]
ResNet 34	21.53	5.6	~50M	3.6G
ResNet 152	19.38	4.49	–	11.3G
Inception-v3	19.47	4.48	25M	5.0G
Inception-ResNet-v2	18.7	4.1	–	–

All comparisons are for a single model for top-1 and top-5 classification error rates from the ImageNet 2012 validation dataset. FLOPs refer to the number of multiply-add operations for evaluating an image in the network. All numbers are from the papers [12], [13], [30], [31] released in conjunction with the ILSVRC challenge, or later upon updates of the architectures, unless otherwise cited. Microsoft did not provide the parameter counts for its ResNet 34 and ResNet 152 networks. The ResNet 34 parameter count is an estimate we calculate from Figure 2.6. VGGNet's three fully-connected layers at the end of its network account for about 100M of its parameters. The other networks only apply a single fully-connected layer; thus, they contain significantly fewer parameters. VGGNet and ResNet implemented an average of ten different crops across an image during evaluation whereas the Inception networks average twelve. Empty locations in the table are for figures not provided by model creators and not easily calculated due to model complexity.

is possible to train models from scratch on new classification categories. Realistically, acquiring and labeling a sufficient volume of training data for each new problem proves inefficient and severely limits the tasks for which users can apply these powerful models. Thankfully, another training method called *transfer learning* [38] exists. This method capitalizes on the models' several levels of abstraction, as discussed in Section 2.3.2. Since the lowest-level-convolutional layers learn to extract features like edges and color patterns that apply to any category of classification, users do not need to retrain these layers for each new task. Similarly, as noted in [38], the middle and higher levels of abstractions also prove useful in most tasks for networks trained on diverse classification problems like ILSVRC. Thus, it is possible to simply replace the last level of abstraction, the categories themselves. Starting with a network trained on a large image set like ImageNet, one can simply retrain the last fully-connected layer of network. See the darker purple oval second from the right of Figure 2.8 to gain an appreciation for all of the layers to the left of this purple oval that we reuse with transfer learning. With this method, it is possible to reduce training time

from days to minutes, and the number of images required from tens of thousands to only hundreds, while still achieving state-of-the-art performance.

With an understanding of supportive policy for acquiring TensorFlow in the DOD, previous NPS experiments and image sets that serve as potential application areas, and state-of-the-art models and training methods available through the software library, it is now possible to develop experiments to provide empirical evidence of TensorFlow's suitability for DOD specific applications.

CHAPTER 3:
Methodology

This chapter presents the logic behind our experimental design as we pursue answers to our research questions. Through our experiments we explore Google's TensorFlow software framework and evaluate its usefulness across the DOD. To accomplish this, we compare the performance of algorithms implemented with the software to three previously employed machine learning methods and experiments conducted at NPS. Here, we first provide the scope of our research since we do not test every potential application of TensorFlow. Then we discuss the reasoning behind our design of three experiments within our scope. Overall, this chapter justifies our experimental design and our selection of applications of the software we test.

3.1 Scope

TensorFlow is a significant software library with more functions and applications than we are able to test. Our evaluation of TensorFlow focuses on object detection as a starting point to make recommendations for future DOD research towards the implementation of open-source software for other machine learning tasks. To further scope the problem, this study will not:

- Conduct an analysis of the other available open-source frameworks
- Implement new algorithms for image classification
- Complete testing of these algorithms in operational environments
- Test all TensorFlow use cases (i.e., speech and text recognition and prediction)

The purpose remains to provide a logical and empirically based recommendation for the DOD to pursue open-source deep learning software solutions in the future. In setting out to achieve this end and answer our research questions, three lines of effort for applying TensorFlow arise within our scope: employing TensorFlow-based algorithms without modification when an existing category applies to a current object; repurposing such algorithms for DOD-specific use cases through transfer learning when 50 or more training images exist for an object; and training proven algorithms from scratch to build new object de-

tectors when many thousand training examples are available. We select these applications of TensorFlow to provide a sufficient survey of the software's utility with respect to DOD requirements. Described in detail in upcoming sections, we find reasons for applying one of these lines of efforts to each image set from the three NPS theses introduced in Chapter 2.

3.2 Employing TensorFlow-Based Algorithms without Modification

The trained Inception models available through TensorFlow already afford cutting-edge classification performance on the 1,000 categories from ILSVRC 2012. We sought to apply these algorithms without modification to a DOD media analysis problem. Jones' [15] AK-47 image set met the necessary criteria as two categories from the ILSVRC 2012 training set appeared useful for detecting these weapons in images: the *rifle* and the *assault rifle, assault gun (AR)* categories. The presence of these relevant categories allows us to simply run Jones' [15] test set images through the algorithms to establish their ability to identify weapons, like AK-47s, in intelligence-related images. Since TensorFlow provides access to multiple Inception models, each one reportedly improving performance on the ILSVRC 2012 validation set [12], [13], we choose to repeat the evaluation for each version. This repetition captures a snapshot of Google's improvement of the algorithm as an AK-47 detector over the five iterations. After seeing high performance on Jones' [15] test set, we then decided to conduct tests of the best performing model on a collection of more difficult AK-47 images and negative examples to further stretch the algorithm. Finally, one of our research questions asks how deep CNNs implemented in TensorFlow perform in resource-constrained environments. In pursuit of this answer we chose to evaluate the five Inception models with an ideal allocation of compute resources and also with limited resources comparable to those available in military-deployed environments. The overall purpose of this experiment is to prove the power of open-source algorithms available through TensorFlow without requiring modification and to show the promise of future performance improvements based on past revisions.

3.3 Repurposing Available Algorithms with Transfer Learning

For a second image set, the collection of ship images from Camp [16], we repurpose a single Inception model using transfer learning to build a binary classifier for ship detection. The major factor driving our decision to apply transfer learning to Camp's [16] image set was the limited number of training examples in the dataset. Also, ship-related categories in the ILSVRC 2012 image set produce a confidence that the previous-learned features are relevant for distinguishing Camp's [16] *ship* or *no ship* categories. We elect to retrain a single Inception model for the sake of simplicity. For this experiment, we select the Inception-v3 model due to its balance of performance and also computational efficiency (see Table 4.3). Since the related operational scenario for a ship detector in a surface-based UAS would likely mandate constrained compute resources, the selection of Inception-v3 is logical. Further, Google provides a tutorial and script, cited later in Section 4.2.4, that simplifies the process of conducting the experiment. In completing this second experiment, we strive to prove the applicability of TensorFlow-based object detectors for problems with limited training examples and in resource-constrained systems.

3.4 Training a Deep CNN from Scratch

The third and final experiment trains three publicly released deep CNNs from scratch and tests their performance in identifying screenshots from Sharpe's [17] image set. Since deep algorithms contain significantly more learnable parameters than other machine learning algorithms, we understood that our three previous image sets did not contain enough examples for both training and evaluation. Hence, Sharpe's [17] image set is the focus of the third test of TensorFlow as her subject, screenshots, proves the simplest of the three to acquire. We had to assemble a new training set for this experiment. By doing so, we explore TensorFlow's utility in applying a model tailored down to the lowest-level-feature extractors to meet defense-specific media analysis needs. The goal of our final experiment is to prove that when an organization in the DOD possesses a substantial labeled image set, they can train a deep CNN from scratch with open-source software and algorithms.

This concludes the logic behind our experimental design. Chapter 4 focuses on providing specific details and processes we execute to perform these three experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Experiments

This chapter presents the details and processes for employing deep CNNs in TensorFlow on the three previous image sets introduced in Chapter 2. For each experiment, we discuss the image sets and steps for training and evaluating the models along with the performance measures applied to produce our results for Chapter 5. Throughout this chapter, it is worth remembering the goal of our experiments: to provide an empirical assessment of TensorFlow's applicability in the DOD through completing experiments that simulate potential use cases.

4.1 AK-47 Detection

This section describes our experiment applying the five Inception models available in TensorFlow to Jones' [15] and other AK-47 test sets. We first describe details for all of the image sets tested. Then, we cover the training of the Inception models conducted by Google researchers and describe our process for evaluating the models. Finally, we introduce our performance metrics for all of our experiments and compare them to specific metrics developed by Jones [15].

4.1.1 AK-47 Related Image Sets

As a starting point, we introduce the images employed for training and evaluating the Inception models (see Table 4.1). Google trains these algorithms to classify images into 1,000 categories from the ILSVRC 2012 training set. In order to determine if an AK-47 exists in an image we reference the *AR* category and the *rifle* category. The table combines the remaining 998 *other categories* Google employed to train the Inception models. We evaluate the models' performance on three sources of test images. The first is from Jones' [15] experiment. The second, *AK-47 Internet Images*, is a collection of AK-47 images from the NPS Vision Lab database in dynamic settings, like urban combat environments. The *other* category from this set possess images with groups of people in congested environments: in living rooms, large groups posing for photographs, and crowds. In the third test set *Rifle-Like Images*, we gathered 100 images from the internet with people

carrying objects like paintball guns, lawn and power tools, and sports equipment. We chose these images to ensure the algorithm did not show bias towards depictions of people holding everyday objects. We include the last two image sets for testing in an attempt to stretch the algorithm further than the more generic video stills provided by Jones [15].

Table 4.1. Training and test sets for the AK-47 experiment.

Training Sets	**Category**	**Number of Images**
ImageNet [1]	assault rifle, assault gun (AR)	1,172
	rifle	1,475
	other categories	1,278,520
Test Sets		
Jones [15]	AK-47	687
	other	7,045
AK-47 Internet Images	AK-47	107
	other	143
Rifle-Like Images	other	100

4.1.2 Inception Training

For this experiment we rely on training conducted and documented by Google researchers for the following Inception models: v1 [42]; v2 and v3 [12]; and v4 and ResNet v2 [13]. We conduct no further training or manipulation of these models for reasons stated previously. The trained models, implemented in the TensorFlow-Slim [44] library, are available for download [45].

4.1.3 Inception Models Evaluation

After downloading the pretrained models, we modify scripts provided from TensorFlow tutorials [46] to conduct our experiment. The script originally took a single image as input and produced the top-five classification scores. Our modifications allowed us to run all test images from a file directory through the models one image at a time and save all 1,000 classification scores. We ran the experiments with the Python 2.7.12 interpreter and TensorFlow version 0.11.0rc2. We repeat the following process for each of the five Inception versions. The software first reads an image from a file and decodes it into a bytecode format

that the Inception models require for input. It also conducts a bilinear rescaling and then a central crop on the image to reduce it to the required 299 by 299 pixel input size. After preprocessing, we feed the image through the model while keeping scores from the three test sets separate. We then save the model's output per image: a score for each of the 1,000 ILSVRC 2012 categories (plus a background category score that we ignore) with the sum of the scores equaling one. Think of these scores as a probability of each object existing in an image according to the model.

We repeat our evaluations of the test images twice to provide time estimates for classifying images with ideal hardware and in a resource-constrained environment. Table 4.2 displays the hardware devices and specifications employed throughout all three of our experiments. Using the TITAN Black GPU, we provide times for processing the AK-47 test set as single images through the Inception models in Table 4.3. For the constrained tests, we engage a portion of the i7-5820K CPU's resources through a virtualized Docker [47] container with only two cores of the processor and 4GB of RAM. The times represented in the table measure the entire process to accomplish the image classification for all of the three test sets: first the software restores the trained model version to memory; then, it runs a single image at a time through the model, including preprocessing; next, the script saves image scores in a comma-separated value file by directory; and finally it removes the model from memory. The table also presents the average time per image by dividing the total time by the number of images in all three test sets. While not a measure of the time it takes to feed a single image through these models, our measurement provides an estimate for how long it will take for the algorithm to classify a group of images and present them for input to other software modules or store them in a database. There are possible time savings, which we will present in Section 4.3, if users store the images in a format that allows the model to process them in groups called *batches*, typically ranging from sizes of 30 to 100 images.

4.1.4 Performance Metrics

Having gathered the raw output scores by category for each of the models, we compare them to the true labels of the test images to grade the classification performance of the Inception versions. First, we separate the *AR* category score and the *rifle* category score from the raw output scores. Then, we consider individually the *AR* score and the *rifle* score. To generate a third score for evaluation, we add the previous two category scores together. Finally, we

Table 4.2. Hardware employed for experiments.

GPUs	Clock Speed	Cores	Memory(GB)	Memory Interface
TITAN Black	980 MHz	2,880	6.144	GDDR5
1080 Ti	1.582 GHz	3,584	11	GDDR5X
CPUs				
i7-5820K	3.30 GHz	12	32	DDR4-2400
i5-3317U	1.7 GHz	2	4	DDR3

Both GPUs are from the NVIDIA GeForce GTX series. The TITAN Black resides in a compute node on the NPS Hamming Supercomputer. The 1080 Ti resides in a personal desktop computer. Both CPUs are from the Intel Core series. The i7-5820K is in the same desktop computer as the 1080 Ti GPU. The i5-3317U exists in a laptop computer and provides for an evaluation of TensorFlow in a resource-constrained environment.

Table 4.3. Inception models' image classification times.

Inception Version	Total Time	Time per Image	Images / sec
GPU			
v1	141.7	0.0175	57.0
v2	174.2	0.0216	46.4
v3	276.6	0.0342	29.2
v4	485.4	0.0601	16.6
ResNet v2	565.7	0.0700	14.3
CPU			
v1	968.1	0.120	8.3
v2	1128.8	0.140	7.2
v3	2979.9	0.369	2.7
v4	5711.0	0.707	1.4
ResNet v2	6071.9	0.751	1.3

All times are in seconds. We ran the image evaluation script on both the TITAN Black GPU and a portion of the i7-5820K CPU's resources through a virtualized Docker [47] container with only two cores of the processor and 4GB of RAM. See Table 4.2 for hardware details. We ran a total of 8,082 images through each model to generate the time values displayed in this table.

compare these three scores to a sliding threshold of 1,000 equally spaced values between zero and one. At each, a score above the threshold value indicates a positive classification

by the model. We then count the true positives, false positives, and true negatives for our labeled test images and use these counts to develop our metrics for the 1,000 thresholds.

From these counts, we calculate the *recall*, *FPR*, and *F-score* as defined in Equations 4.1, 4.2, and 4.4. We derive *precision* (defined in Equation 4.3) as a necessary intermediate step to provide the F-score. As shown in Equation 4.4, the *F-score* metric is the harmonic mean of both precision and recall which we provide to identify an optimal threshold for classifier performance. We set β equal to one to attribute equal importance to both precision and recall. Finally, we define *accuracy* here for completeness but only calculate it to compare model performance during training for the final two experiments. Plotting the recall versus FPR values at all of the 1,000 thresholds for the three different scores produces our ROC curves presented in Chapter 5.

We employ the following abbreviations in the equations in this section:

- TP = True positives
- TN = True negatives
- FP = False positives
- PE = Total number of positive examples
- NE = Total number of negative examples

Equations for recall, FPR, accuracy, and precision [48] and F-score [49]:

$$\text{recall} = \frac{TP}{PE} \tag{4.1}$$

$$\text{FPR} = \frac{FP}{NE} \tag{4.2}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{4.3}$$

$$\text{F-score} = \frac{(\beta^2 + 1) * \text{precision} * \text{recall}}{\beta^2 * \text{precision} + \text{recall}} \tag{4.4}$$

$$\text{accuracy} = \frac{TP + TN}{PE + NE} \tag{4.5}$$

4.1.5 AK-47 Experiment-Specific Performance Metrics

We must note that our results do not directly compare to Jones' [15] results. While we both present ROC curves as a performance measure, Jones' [15] algorithm evaluates an image several times before providing a count of AK-47s in the image instead of an overall image classification. Thus, his definition of recall and FPR reproduced in equations 4.6 and 4.7 vary from the standard definitions. Still, we present a general comparison of performance in Chapter 5.

Equations from Jones' [15] thesis:

$$\text{recall} = \frac{NumberOfWeaponsDetectedInSet}{TotalWeaponsInSet} \tag{4.6}$$

$$\text{FPR} = \frac{NumberOfFalsePositivesInImageSet}{TotalAreasChecked} \tag{4.7}$$

4.2 Ship Detection

The second experiment sets out to apply deep CNNs to a DOD image analysis task, ship detection, even though few training images are available. This section describes the image sets, image distortions, optimizations to reduce training time, and the training process for reapplying the pretrained Inception-v3 model from the previous experiment. Through these methods we replace the model's 1,000 possible categories to produce only two classification scores; *ship* or *no ship*. The evaluation process and performance metrics remain the same as those presented previously.

4.2.1 Ship Related Image Sets

As described for the AK-47 experiment, we start with an Inception-v3 model pretrained on the ILSVRC 2012 training set. There are several relevant training categories from this

image set for ship classification. These categories suggest the feature extractors from the original Inception model, that will remain in our new *ship* or *no ship* classifier, will be successful; see Table 4.4 for a selection. Most of the images from Camp's [16] thesis are close-ups of ships or contain water scenes with coastline, mountains, ice, or wildlife and the high performance of our first retrained model inspired us to acquire some more challenging examples for testing. Thus, we built the *Ship Internet Images* set by collecting images of 67 ships on the horizon, with more unique shapes, in front of sunsets, in fog, and other more difficult settings. We also collected 86 negative images of close-ups of objects like trash, cargo containers, and debris floating in the water. Prioritizing a more rigorous test of the model, we divided this set into training and test examples by only transferring seven ship examples and 26 negative examples into the training set. After observing a high FPR on the *Ship Internet Images* set by an algorithm retrained with only Camp's [16] training set, we sought to show that the Inception-v3 model was capable of classifying these more difficult images by developing a small training set relating to the *Ship Internet Images* negative examples. With the easy-to-find example images already in the test set, we built a training set without requiring all negative examples to be water scenes. Specifically, we located images of trash in landfills and other settings in hopes of teaching the algorithm that trash in the water was not a ship without requiring more examples of trash floating in the water. With this mindset, we found 226 additional negative training examples for the *Ship Internet Images* training set by including images of trash and shipping containers not in the water, as well as buoys, oil rigs, sunsets over water, and ocean wildlife.

4.2.2 Image Distortions

Using flags available in the retraining script, we applied several different distortions to the images during training. Table 4.5 presents the available distortions. We vary the combinations of these operations during the training phase to permit an evaluation of their affect on model performance. While these distortions can lead to increased classifier performance by artificially increasing the size of training set, engaging any one of them significantly slows the retraining process for reasons we will discuss in the next section.

Table 4.4. Training and test sets for the ship detection experiment.

Training Sets	Category	Number of Images
ImageNet [1]	container ship, . . .	1,613
	speedboat	1,153
	pirate, pirate ship	634
	seashore, coast, . . .	2,382
	other categories	1,275,385
Camp [16]	ship	110
	no ship	97
Ship Internet Images	ship	7
	no ship	226
Test Sets		
Camp [16]	ship	405 (8x)
	no ship	99* (8x)
Ship Internet Images	ship	60
	no ship	60

This table is not all-inclusive of useful ImageNet categories as we omitted some that contain smaller water craft. *Note: this number is one less than the reported 100 images from Chapter 2. We found one repeated image in both the training and test set from Camp's thesis and thus removed it from the test set. Camp's test set had all images scaled at 5, 10, 15, 20, 25, 50, 75, and full scales to simulate classifier performance on ships further from a sensor or on the horizon.

4.2.3 Bottleneck Calculations

As mentioned previously, retraining only the last fully-connected layer of a deep CNN can significantly reduce the time required to apply an existing algorithm to a new task. The program provided for retraining the Inception-v3 model achieves a large portion of this efficiency by creating *bottleneck files*. These files are essentially the model's output of the layer immediately before the last fully-connected layer; the only layer we update during training. Running each image through about 98% of the network only once, and storing the output as bottleneck files for later training steps, we avoid the additional operations required to feed an image through the entire network hundreds of times [50]. A downside of the operations discussed in Section 4.2.2 is that each distorted image is a new image that we must run through the entire network, eliminating this efficiency and significantly increasing training time. Table 4.6 illustrates this slow-down when retraining the Inception-v3 model under resource constraints, such as on a laptop in a deployed environment. Still, even with

Table 4.5. Distortions available in the retraining script from TensorFlow.

Distortion Name	**Effect**
Random Crop	Accepts a percentage (*RCP*). Randomly picks a center for the first crop in the image and maintains aspect ratio while keeping a portion of the image that has a width of the original image minus the *RCP*. *RCP=10* would produce an image whose width was 90% of the original width.
Random Scaling	Accepts a percentage (*RSP*). First conducts a central crop to create an image whose scaled width (*sw*) is chosen by a random variable (r) in a range from original width minus *RSP* to the original width (w) as follows: $sw = w - (\frac{RSP}{100} * w) * r$. This distortion then employs bilinear interpolation to resize the cropped image by a random percentage between zero and the *RSP*.
Random Brightness	Accepts a percentage (*RBP*). Using a random uniform distribution, selects a number between $1 \pm \frac{RBP}{100}$ to multiply with the original image pixel values.
Random Flip	Accepts a Boolean. If the Boolean equals true, randomly flips 50% of images horizontally left to right during training.

These distortions help to ensure the model is invariant to changes in the depiction of an object in an image. Rather than requiring unique examples of an object at different scales, horizontal orientations, brightness, and occlusions, these functions instead modify existing images. This effectively increases the size of the training set [50]. These distortions are specific to the Inception-v3 retraining script [51] available through TensorFlow.

distortions, transfer learning provides efficiencies from not requiring gradients to pass all the way back through a network during backpropagation and weight updates. The training times presented show it is possible for DOD members to repurpose pretrained models to unit specific tasks in environments with limited resources.

4.2.4 Retraining Inception-v3 via Transfer Learning

This section describes the process for retraining the Inception-v3 model to classify images as *ship* or *no ship*. We employ the 1080 Ti GPU from Table 4.2 for all of the experiments in this section. We follow a simple training tutorial [50] and script [51] for guiding this process. We first retrain the model with only Camp's training set, no distortions, and the following hyperparameters:

Table 4.6. Retraining times for Inception-v3 in a resource-constrained environment.

Distortions	Training Steps	Training Time
None, default	4,000	5m
None, default	8,000	11m
Random Scale (10%)	4,000	1d 22h 54m
All(10%) and Flip(T)	4,000	1d 23h 17m
All(10%) and Flip(T)	8,000	3d 22h 30m

This table depicts slow down in network retraining introduced by distortions. We conducted these tests on a personal laptop running 64-bit Linux Ubuntu 16.04, with the i5 CPU from Table 4.2. We ran TensorFlow version 0.8.0 in a virtualized Docker container having access to all system resources to produce these numbers, following the steps described in [50]. This method has a one-time cost of approximately 25 minutes to compile the initial model to maximize training and evaluation speed on our specific CPU. We complete all other iterations with the same model with default settings and change the distortion flags from Table 4.5 to the values in parenthesis. The training set included about 2,000 images divided into two categories and the batch size per step was 100 images.

- *batch size* = 100
- *training steps* = 1,000
- *learning rate* = 0.01
- *validation percentage* = 10%

We leave the *batch size*, the number of images fed through the model per training step before updating the model's learned weights, and learning rate as the default recommended values provided with the script. We limit the training steps to 1,000 due a larger batch size and smaller number of training examples. In this case the model trains on each image 535 times, or completes 535 *epochs* of training. The validation percentage flag provides how many images to set aside from the training set to evaluate performance of the model throughout training. The script selects the validation images based off of a hash of their filenames; they remain consistent across runs. Table 4.7 provides the training and validation image counts for our runs. The size of our validation set is a source for concern; however, when working with a limited training set it remains necessary to keep as many images as possible for training. For each of the runs, TensorFlow logs important training information that users can access via TensorBoard [52].

Table 4.7. Summary of ship detector training iterations.

Iteration Name	**Train (+/-)**	**Valid. (+/-)**	**Distortions(Value)**
ship_images_orig	99 / 88	11 / 9	None
ship_images_added_default	106 / 323	11 / 32	None
ship_images_added_bright10	106 / 323	11 / 32	Random Bright. (10%)
ship_images_added_crop10	106 / 323	11 / 32	Random Crop (10%)
ship_images_added_scale10	106 / 323	11 / 32	Random Scale (10%)
ship_images_added_flip	106 / 323	11 / 32	Random Flip (T)
ship_images_added_bsc10flip	106 / 323	11 / 32	All Distortions (10%), Random Flip (T)

The train and validation numbers represent the number of *ship* (+) and *no ship* (-) examples available during training. For the last model, we set random brightness, scale, and crop to ten percent and also the random flip flag to true in order to observe the affect on performance of combining the distortions.

Figure 4.1 displays the performance of our first model using only Camp's image set; it classified the validation set perfectly. Due to high training and validation accuracy, we did not train additional models with only these images. Instead, we added our *Ship Internet Images* training set to Camp's and repeated the training process. We apply the same defaults as above, but vary the distortions as depicted in Table 4.7.

From Figure 4.2, a screen capture of TensorBoard, we observe the validation accuracy to select the best model for another round of training. Examining the plot, we see the distortions do not improve the performance. Thinking about the Inception-v3 model's initial training, one can derive a logical reason for the distortions' apparent ineffectiveness. Google originally trained the lower layers reading the raw pixel inputs on distorted images. Thus, the low-level feature extractors that distortions affect most are already invariant to them. Realizing this, we did not employ them for any further retraining iterations.

Figure 4.2 presents classification accuracy on the validation set plotted against the number of training steps completed. The lighter lines depict raw accuracy measures, recorded every 10 training steps, whereas the darker lines represent a smoothed value from these measures. The model without any distortions slightly out performs the others at its final iteration. Selecting this as the best model to train for another round, we decreased the *learning rate* to 0.001 and increased the *training steps* to 4,000. Figure 4.3 presents a comparison of

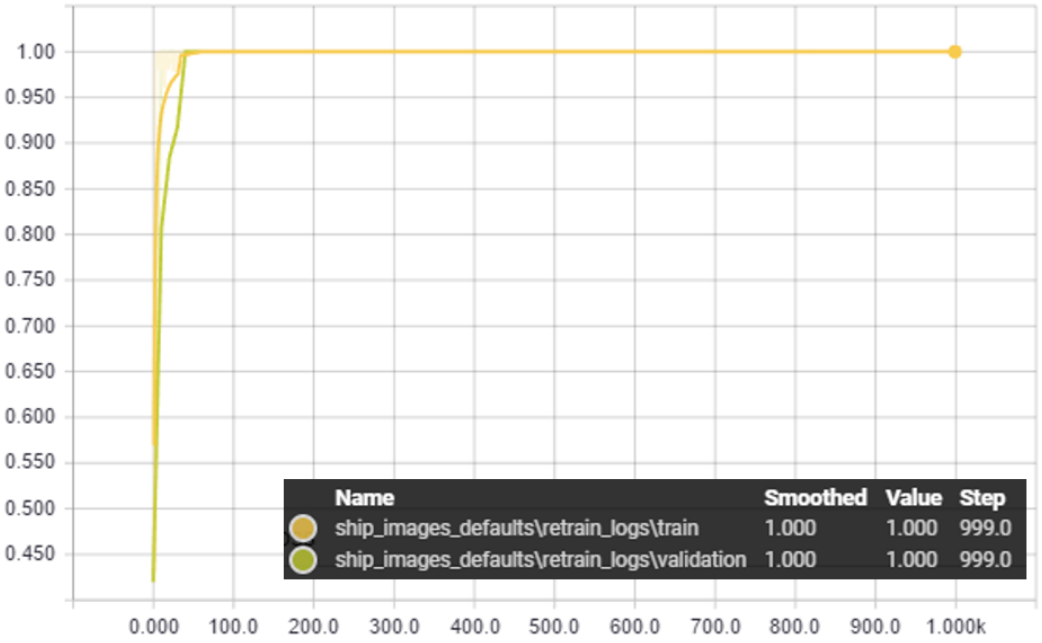


Figure 4.1. Ship detector training and validation accuracy vs. training step with only Camp's [16] images from TensorBoard. This plot and legend are screen captures from the web-browser based TensorBoard application that is a part of TensorFlow. The interactive plot allows a user to slide his mouse along the training steps to observe values stored at each step. The legend displays the final values for each model. We capitalize on our goal to present the functionality of TensorFlow to avoid exporting this data via comma-separated values and generating our own plots; similar figures throughout this thesis are also from TensorBoard.

the validation accuracy for our top-performing initial model (*ship_images_added_defaults*) and its second-round configurations (*ship_images_added_defaults_rnd_2*).

As is evident in Figure 4.3, validation accuracy remains roughly constant after 1,000 training steps. This observation, combined with our desire to avoid overfitting due to the fact that we conduct training with a relatively small data set, encourages a selection of the original model completing only 1,000 training steps. We look elsewhere for confirmation of this intuition. The retraining script also logs the softmax layer's *cross-entropy loss*, a measure of the difference between the model's predicted category scores for an image compared to the labeled true category, which training strives to minimize. Observing Figure 4.4, the shallow descent and higher final cross-entropy loss of the second round model suggest we

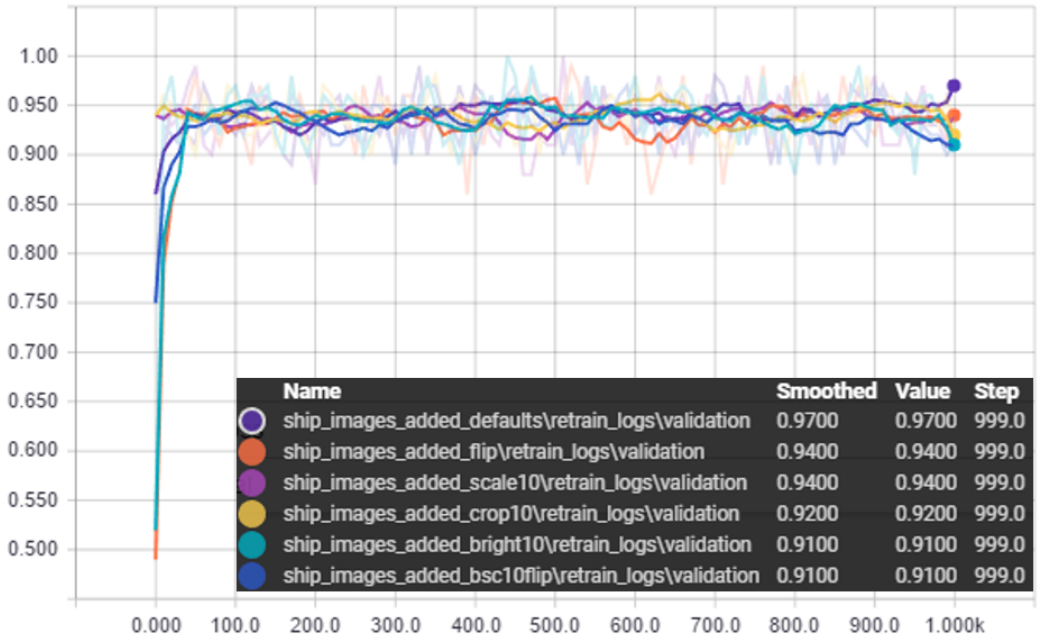


Figure 4.2. Ship detector validation accuracy vs. training step for first round of training. This figure displays the ineffectiveness of the distortions (described in Table 4.7) in improving accuracy on the validation set. The top-performing model (*ship_images_added_defaults*), did not employ any distortions and we carry it forward for further comparisons in Figure 4.4.

set the learning rate too low. The combination of these two pieces of evidence provide confidence in selecting the default model (*ship_images_added_defaults*) trained for 1,000 steps to evaluate the test sets against.

4.2.5 Evaluating Retrained Inception-v3 on Ship Classification

We evaluate the retrained model with a nearly identical process to the one described in Section 4.1.3. We maintain the separation of Camp's [16] test set with our *Ship Internet Images* test set and redo this evaluation process to store the scores for *ship* and *no ship* returned for each image. We do not apply distortions during the evaluation of the test images besides rescaling and cropping to reduce the image to the required Inception-v3 model input of 299 by 299 pixels.

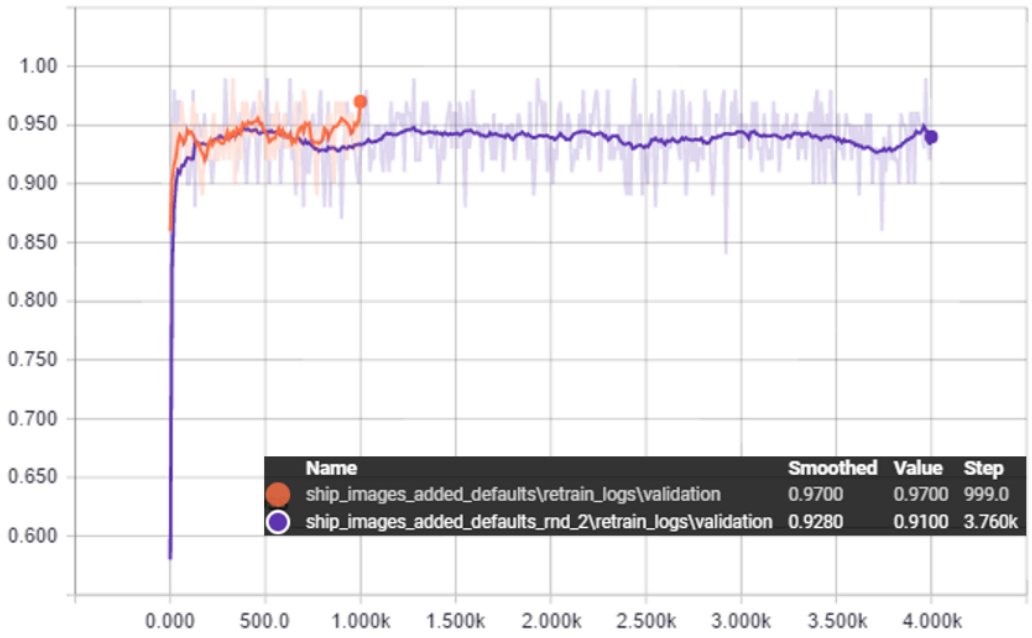


Figure 4.3. Ship detector validation accuracy vs. training step for second round of training. The *ship_images_added_defaults* model is the top-performing model from Figure 4.3. The second round model (*ship_images_added_defaults_rnd_2*) is the same model from the first round but we train the second model with a smaller learning rate (0.001 instead of 0.01) and for an extra 3,000 training steps.

4.2.6 Ship Experiment Performance Metrics

Camp does not directly describe his calculations for recall and precision; we assume he employs the standard equations presented in Section 4.1.4. He does however present his results by ROC curves. We repeat our preceding method to develop our ROC curves to allow a direct comparison of performance to Camp's curves in Chapter 5.

4.3 Screenshot Detection in Images

Following suit, this section will discuss the image set, training and evaluation processes, and metrics for comparing performance between modern algorithms built in TensorFlow and those employed by Sharpe [17] in order to detect screenshots in images.

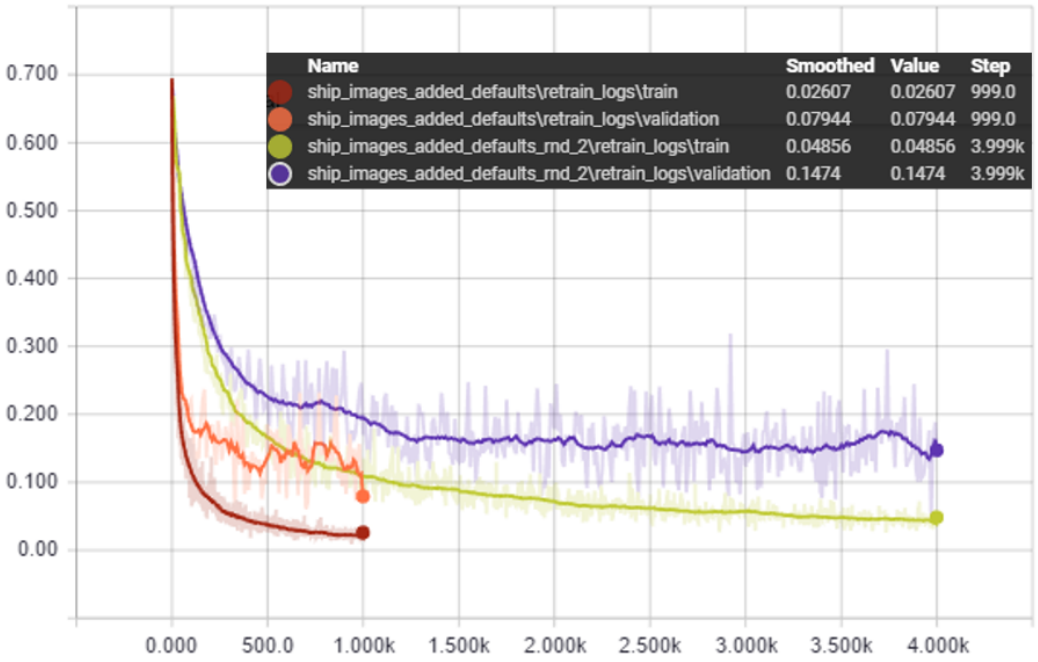


Figure 4.4. Training and validation cross-entropy vs. training step for both rounds of training.

4.3.1 Screenshot Image Sets

Training a deep CNN from scratch demands significantly more labeled examples than transfer learning. As presented in Chapter 2, Sharpe [17] gathered over 6,000 images for her experiment. While substantial, her image set pales in comparison to those as large as the ImageNet dataset. Also, because her algorithms employ engineered feature extractors instead of learning them from scratch, she only required ten percent of these images for training. Needing to test our models against at least 4,800 of her images for a true comparison, we set out to find other examples for training. First, we select ten ImageNet categories representative of objects or scenes in Sharpe's [17] negative image set. Next, converting 192 videos from Wikimedia Commons' *Screencast videos* [53] category we capture one frame as a JPEG for every 30 frames of each video. This method produced 11,428 *screenshot* examples. Finally, we wrote a script to automatically capture 177 screenshots while performing various tasks on a personal computer. Within the ImageNet negative examples, there existed approximately 300 digitally augmented images that resembled screenshots. After removing them, we narrow down our training set to 11,605 *screenshot* and 12,789 *other* images. We split these images by reserving about ten percent of them for a validation set to arrive at the

numbers depicted in Table 4.8. Since we did not want to contaminate the validation set, we moved groups of *screenshot* examples derived from the same video into the validation set. This ensured there were no images from the same video in both the training and validation set. By building this *Screenshots* dataset we reserve all of Sharpe's [17] images for testing.

Table 4.8. Training, validation, and test sets for the screenshot experiment.

Training Sets	**Category**	**Number of Images**
ImageNet [1]	house	1,231
	range, mountain range, . . .	2,029
	government building	334
	building, edifice	1,421
	coast	321
	littoral, litoral, . . .	1,932
	furniture, piece of furniture, . . .	2,138
	tree	1,181
	divided highway, dual carriageway	1,213
	street	1,330
Screenshots	screenshot	10,414
	other	11,513
Validation Set		
Screenshots	screenshot	1,191
	other	1,276
Test Set		
Screenshots (Sharpe [17])	screenshot	2,420
	other	3,644

Before beginning training, we check that we did not make the critical mistake of contaminating the test set. To do this we must compare all of the images in the training and validation sets to those in the test sets to be certain no duplicates exist. By employing a hash algorithm on the images' content we can quickly and reliably compare a large number of images. We first enter all of the training and validation images into a hash database by running them through a difference hash programmed in Python [54]. The difference hash tracks gradients in images to match duplicates [55]. We then check all of Sharpe's [17] test images against the hash database. In our case, we found two duplicates from our ImageNet negative examples. We remove these images from our training set to leave the test set

intact. During the image hashing, the difference hash also identified that approximately 5,500 of the 11,605 *screenshot* examples are near duplicates. While unsurprising due to our screenshot generation methods, this is worth noting as a weakness in our dataset. We left the near duplicates as slight variations in the screenshots still provide useful training examples; at worst they cause overfitting of the model, which we can identify during training if the model performs poorly on our validation set. As a final note, we conducted this image hashing process for each of our experiments but save mentioning it until now as it was most important for this large training set gathered from similar sources to the test set.

4.3.2 Training Modern Models to Identify Screenshots

The training process involves two distinct steps: first, import the *Screenshots* training, validation, and test sets into a TensorFlow *Dataset* [56] and second, run the TensorFlow-Slim training functions to initialize and train the three models presented in Chapter 2 from scratch.

After building the *Screenshots* sets described above, we convert the images into a single TensorFlow *Dataset*. We read all of the images from our sets into a TensorFlow file format known as a TensorFlow Record (TFRecord) [57]. This format stores the information from thousands of JPEG files into a handful of TFRecord files that facilitate fast streaming of images into memory without opening thousands of separate files from the disk. While importing Sharpe's [17] grayscale negative example images into this format, we repeat the single channel pixel values three times for each of the red, green, and blue channel to allow our model to receive the expected three channel input size when evaluating these images. We then associate the TFRecord files for each set into a TensorFlow-Slim defined Python object known as a *Dataset*. This allows us to point TensorFlow to a desired image set by passing the *Dataset* name, in our case *Screenshots*, and a split name: training, validation, or test. Using this object, TensorFlow supervises which images get passed to the model for training to ensure the network sees all training examples before repeating the images for subsequent epochs. Our use of TensorFlow's *Dataset* class in our experiments provides access to desired image sets in a manner that prevents contamination among our three splits and facilitates the use of our new dataset with existing TensorFlow training and evaluation functions. Having efficient access to the image content and associated category labels necessary for our experiments, we stand ready to train the models.

Using a TensorFlow-Slim script [58], we supply the *Screenshot Dataset* training split and the following inputs to train the three models covered in Chapter 2:

- *batch size* = 25
- *max number of steps* = 50,000
- *optimizer* = adam
- *model name* = vgg_16, resnet_v2_50, or inception_resnet_v2

With this batch size, number of training steps, and the number of training images available, we train each model for approximately 57 epochs. Table 4.9 depicts the training time for all training iterations of our models. We run the Adam [59] optimizer, with TensorFlow's default configurations, to adjust the *learning rate* throughout training as depicted in Figure 4.5. We train all three models on a Windows 10 desktop employing TensorFlow r1.0.1, Python 3.5.3, and both the 1080 Ti GPU and i7 CPU from Table 4.2. To start the actual training, the CPU loads images from the *Screenshot* Dataset object and then conducts the prescribed preprocessing steps for each model [12], [13], [30], [31]. Provided a properly formatted *Dataset* object, TensorFlow-Slim contains scripts to quickly execute these preprocessing steps and save the correctly-sized images for input to a specific network in a queue in the CPU's memory. The GPU then requests batches of these images for each training step. TensorFlow manages this queue which enables more efficient training of a network on the GPU by reserving its processing power for the forward pass and back propagation of gradients to update the network's weights. The queue also minimizes the time the GPU has to wait on the preprocessing of images. As a separate process on the CPU, we run an evaluation step to capture a model's performance on 500 images from the validation set every 1,000 seconds and then on all validation images at the completion of training. We repeat these steps for each model listed and use TensorBoard to produce the plots in Figure 4.6 and Figure 4.7.

As Figure 4.6 and Figure 4.7 show, the ResNet model achieves the best performance in both training and evaluation on the validation set at the final training iteration. Looking closer at Figure 4.7, at around 35,000 steps both Inception-ResNet-v2 and VGGNet start to perform worse during evaluation. This drop, after a period of about 15,000 training steps with consistent performance in terms of validation accuracy, could mean that the models are overfitting to the training set. Still, both models seem to recover and achieve comparable performance to the ResNet model on the entire validation set after 50,000 steps. Since we

Table 4.9. Training times for two iterations of three screenshot models.

Model	Training Steps	Training Time
Round 1: Defaults		
VGGNet	50,000	8h
ResNet		4h
Inception-ResNet-v2		12h
Round 2		
VGGNet	30,000	5h
ResNet		3h
Inception-ResNet-v2		9h

All times are for training with the 1080 Ti GPU and i7 CPU from Table 4.2. Times are rounded up to the nearest hour.

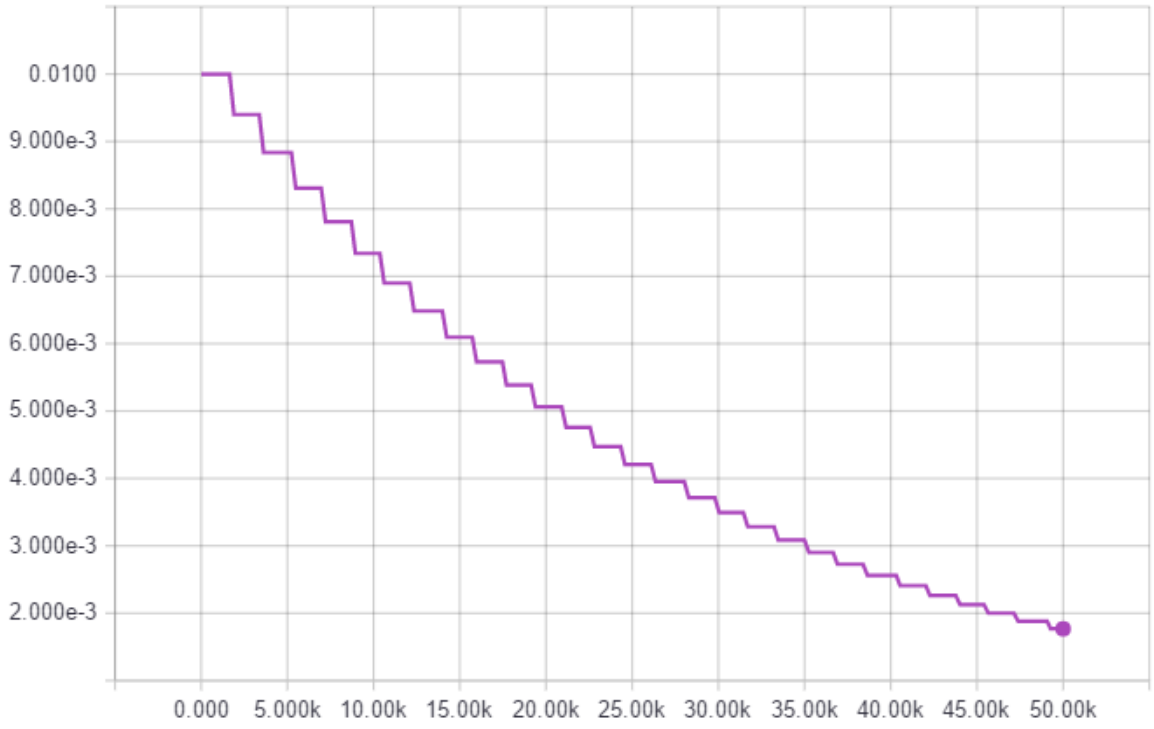


Figure 4.5. Learning rate vs. training steps for all three models. TensorFlow-Slim's training methods adjust the learning rate throughout the training process by employing TensorFlow's implementation of the Adam [59] optimizer. This plot was identical for all three models.

do not get significant separation of performance on the validation set we repeat the training with two adjustments. First, rather than classifying only 500 of the validation images every 1,000 seconds we evaluate the entire set every 2,000 seconds. We increase the interval since this operation slows the training process by consuming most of the CPU's resources for evaluation. Second, we decrease the number of training steps to 30,000 to test our hypothesis of overfitting. Figure 4.8 and Figure 4.9 depict the cross-entropy and validation set accuracy for each model during this second round of training.

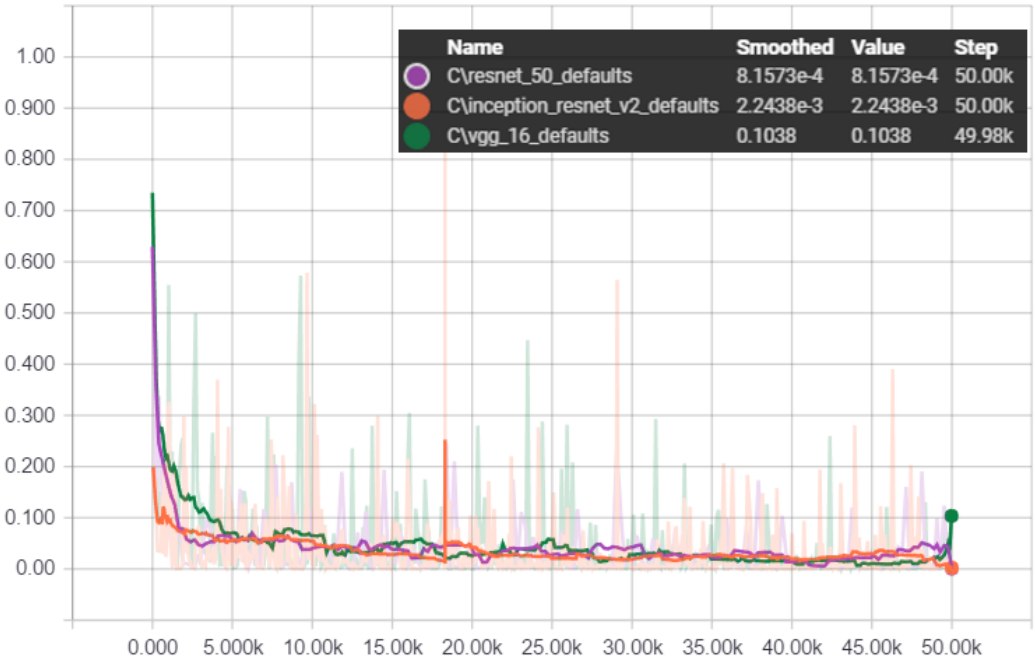


Figure 4.6. Screenshot detector cross-entropy vs. training step for round one.

In round two, the Inception-ResNet-v2 achieves the best performance and scores the highest validation set accuracy out of all models from both rounds of training. However, the ResNet model from round one still scored the lowest cross-entropy loss out of all of the models. Due to these inconsistencies, and some concerns about our newly built validation set accurately representing Sharpe's [17] test set, we carry both iterations of the VGGNet, ResNet, and Inception-ResNet-v2 models forward to compare performance on the *Screenshots* test set. Best practice suggests training hundreds of iterations of these models, adjusting training parameters for each, to achieve maximum performance. We forgo this hyperparameter search

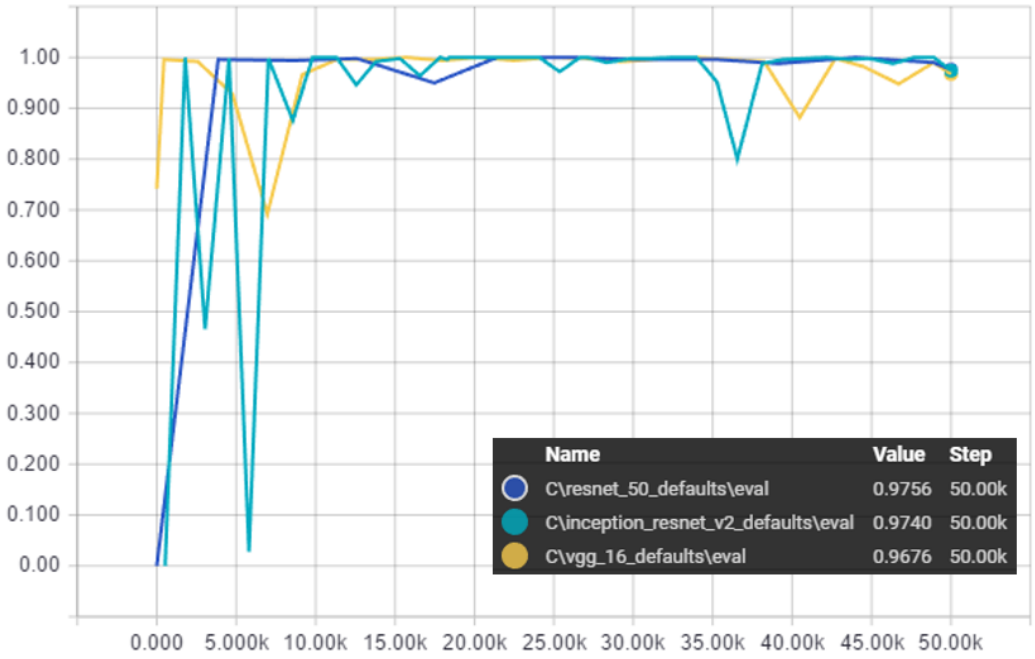


Figure 4.7. Screenshot detector validation set accuracy vs. training step for round one.

due to strong performance on the validation set, time and compute resource constraints, and TensorFlow providing the recommended configurations from each algorithm's creators.

4.3.3 Evaluating Deep CNNs as Screenshot Detectors

The evaluation method for this experiment varies due to our implementation of a TensorFlow-Slim *Dataset*. All software and hardware remains the same from Section 4.3.2. We run our evaluation on all of the available 6,064 archived images from Sharpe's [17] experiments. In contrast, she only evaluates performance through ten-fold cross validation on a selection of 4,800 of these images. With no record of which 4,800 images Sharpe [17] included, we evaluate our best models on all available examples. Similar to training, we engage the CPU to complete model-specific preprocessing steps to generate a queue of images.

For the VGGNet and ResNet models, with pixel values starting as integers between zero and 255, the evaluation preprocessing steps are as follows:

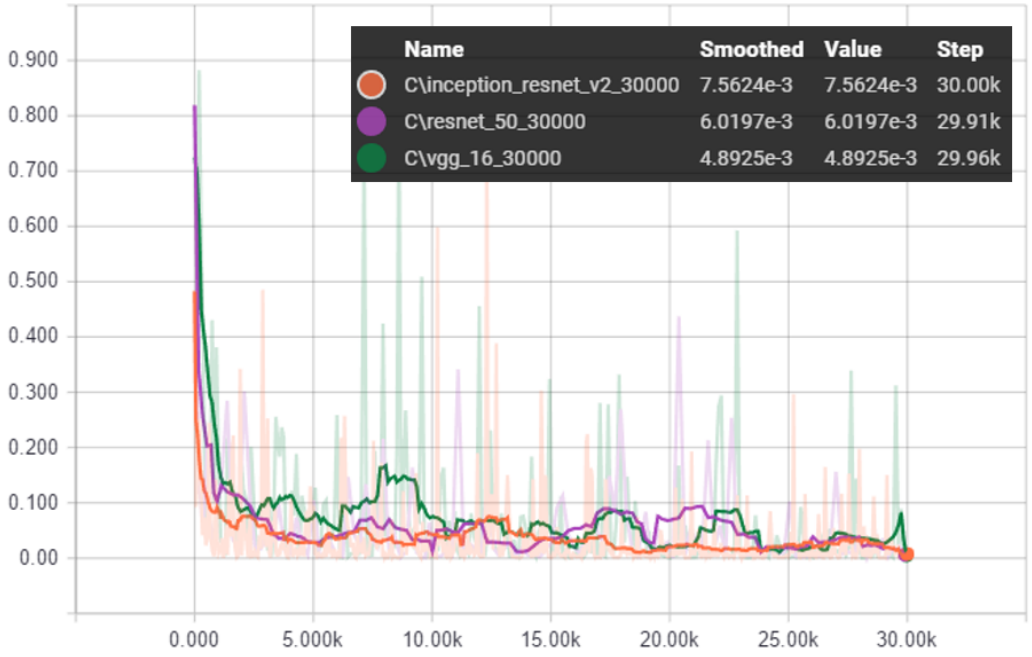


Figure 4.8. Screenshot detector cross-entropy vs. training step for round two.

1. Resize the image so that the smallest side equals 224 pixels.
2. Conduct a central crop to match the longer side to 224 pixels while maintaining the original aspect ratio.
3. Subtract the training set mean red, blue, and green pixel values from each pixel in the image.

For the Inception-ResNet-v2 model, with pixel values starting as 32-bit floats in range between zero and one, here are the steps:

1. Conduct a square-central crop containing 87.5% of the original image area.
2. Execute a bilinear resize on the image to the match a 299 by 299 input size.
3. Subtract 0.5 from all pixel values.
4. Multiply all pixel values by 2.0 to make range between one and negative one.

After preprocessing, TensorFlow sends batches of 100 images to the GPU and runs them through the current model to produce a score for *ship* and *no ship*. Once the 61 batches finish, we store each image filename and associated scores in a comma-separated value file

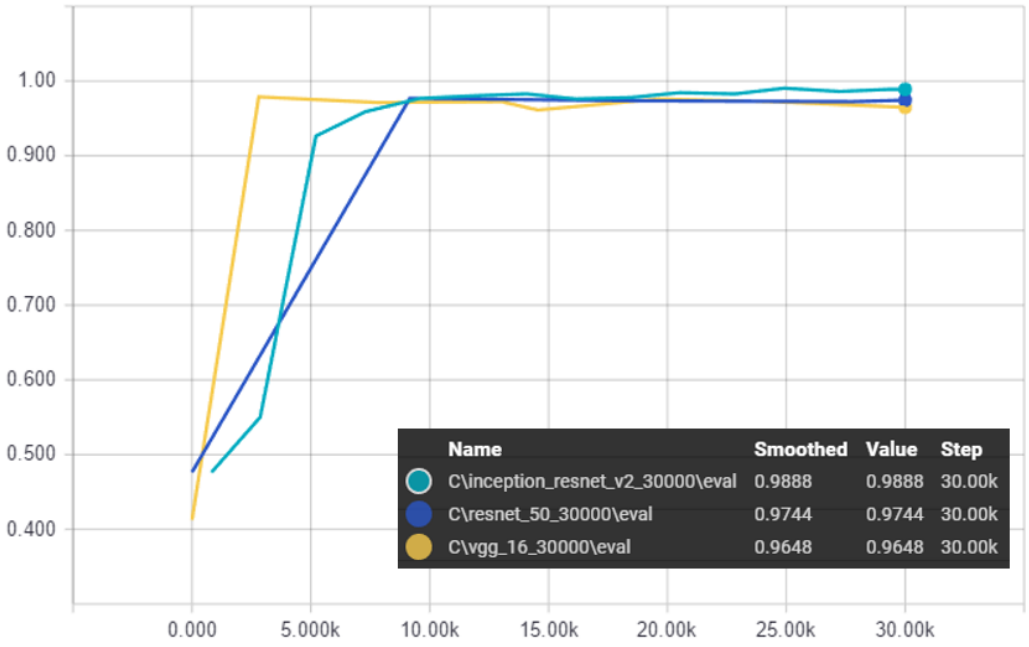


Figure 4.9. Screenshot detector validation set accuracy vs. training step for round two.

for evaluating model performance. Table 4.10 presents the time to evaluate all three models on the test set images.

Table 4.10. Screenshot models' image classification times.

Model	**Total Time**	**Time per Image**	**Images / sec**
GPU			
VGGNet	49.4	0.00814	122
ResNet	38.1	0.00628	159
Inception-ResNet-v2	86.3	0.0142	70

This table displays a measure of the time, in seconds, required to classify images with deep CNNs in TensorFlow when using the TFRecord format and evaluating batches of images simultaneously. We produce these times with the described evaluation process in Section 4.3.3 by employing the 1080 Ti GPU and i7 CPU from Table 4.2 and evaluating 6,064 test images. The 1080 Ti has more resources available than the TITAN Black GPU that produced the times in Table 4.3. Therefore, we cannot compare the Inception-ResNet-v2 model in both tables to attempt to measure specific time savings with the TFRecord format.

4.3.4 Screenshot Performance Metrics

We calculate the same metrics defined in Section 4.1.4 to grade the top model's performance. Sharpe [17] does not provide the equations she implemented for her performance metrics so we assume she follows the standard definitions. Using thre process described previously, we summarize performance in a ROC curve and present the results in Chapter 5.

CHAPTER 5:
Results and Discussion

With our experiments complete, this chapter presents and discusses results with the goal of answering the first four research questions from Section 1.6. Following the order of introduction in Chapter 4, we provide ROC curves and F-scores for the three detectors and compare their classification performance against previous methods. For each experiment, we will examine where the best algorithm fails in an attempt to better understand which features the algorithm learned or missed during training. In Section 5.4 we analyze the training and evaluation times presented in Chapter 4 in order to assess the feasibility of TensorFlow-based image classification in deployed and resource-constrained environments. In summary, we present our empirical measure of TensorFlow in this chapter.

5.1 Performance of the Inception Models as AK-47 Detectors

This section examines the pretrained Inception models' performance as AK-47 detectors. To start, we compare performance across two category scores and their sum: *AR*; *rifle*; and *AR* + *rifle*. Next, we observe the five Inception models' performance on Jones' [15] images. Third, we present performance of the best Inception model, selected by highest F-score on Jones' [15] test set, against the two additional image sets to more vigorously test its classification performance. Finally, we examine the misclassifications on all three test sets to provide further insight into the behavior of the Inception models as AK-47 detectors.

5.1.1 Three Score Comparison

Recording our three separate category-classification scores proves unnecessary. Figure 5.1 displays the ROC curve for the first Inception model on all three test sets. For Inception-v1, the *rifle* category performed its best relative to the *AR* score as an AK-47 detector. Still, it scored a lower F-score than the *AR* category individually and did not improve the F-score when added to the *AR* category. Looking at the latest model, Figure 5.2 depicts the same plot for Inception-ResNet-v2. Here the *rifle* category again performs worse and does not improve F-score when added to the *AR* score. As a result of these findings and

a preference for simplicity, we consider only the *AR* category score for our remaining comparisons. All of the Inception model ROC plots of this three score comparison, and tables containing corresponding performance metrics at the best F-score threshold, are available in Appendix A.1.1.

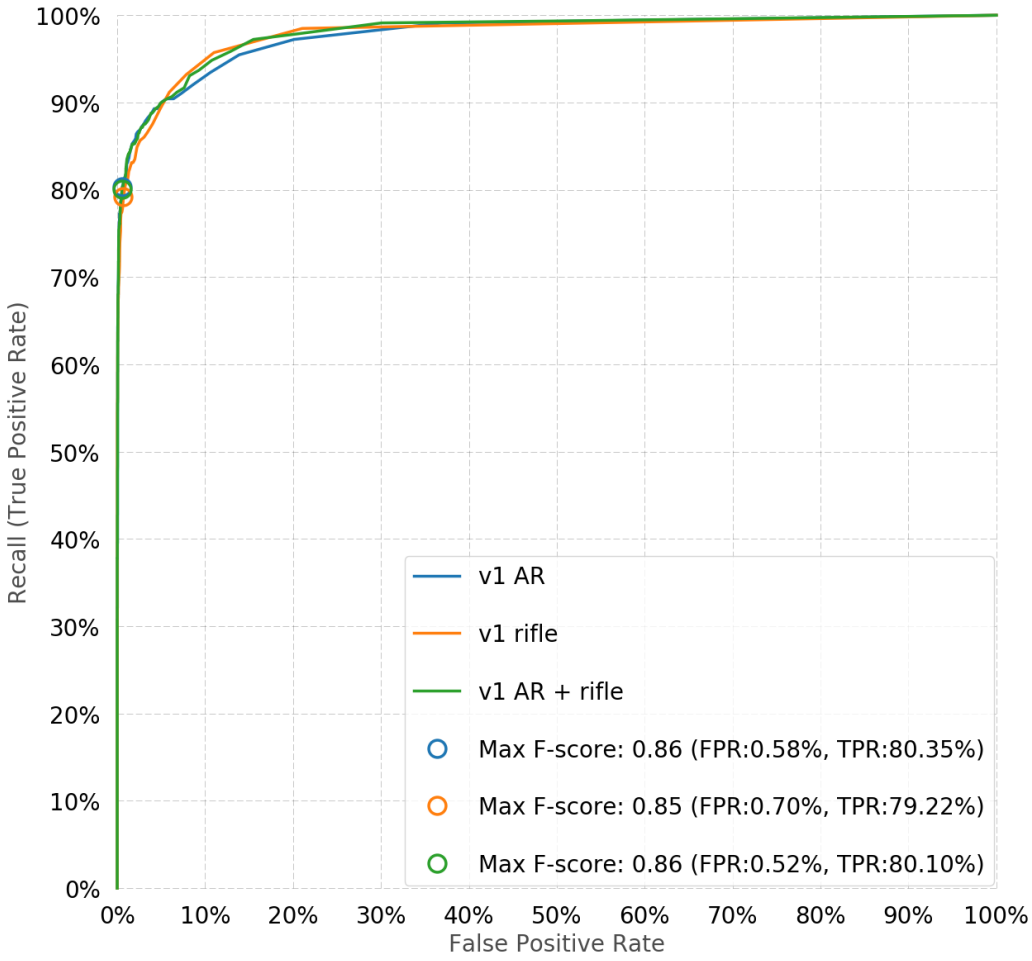


Figure 5.1. Inception-v1 ROC curve for three category scores on all three test sets combined. All of our ROC curves in this section plot the recall versus FPR at the 1,000 score thresholds as described in detail in Section 4.1.4. The lines on these plots connect these values without performing any sort of fitting. The circles plotted on the graph represent the threshold for the highest F-score; we include the recall (TPR) and FPR values generating this F-score in each legend.

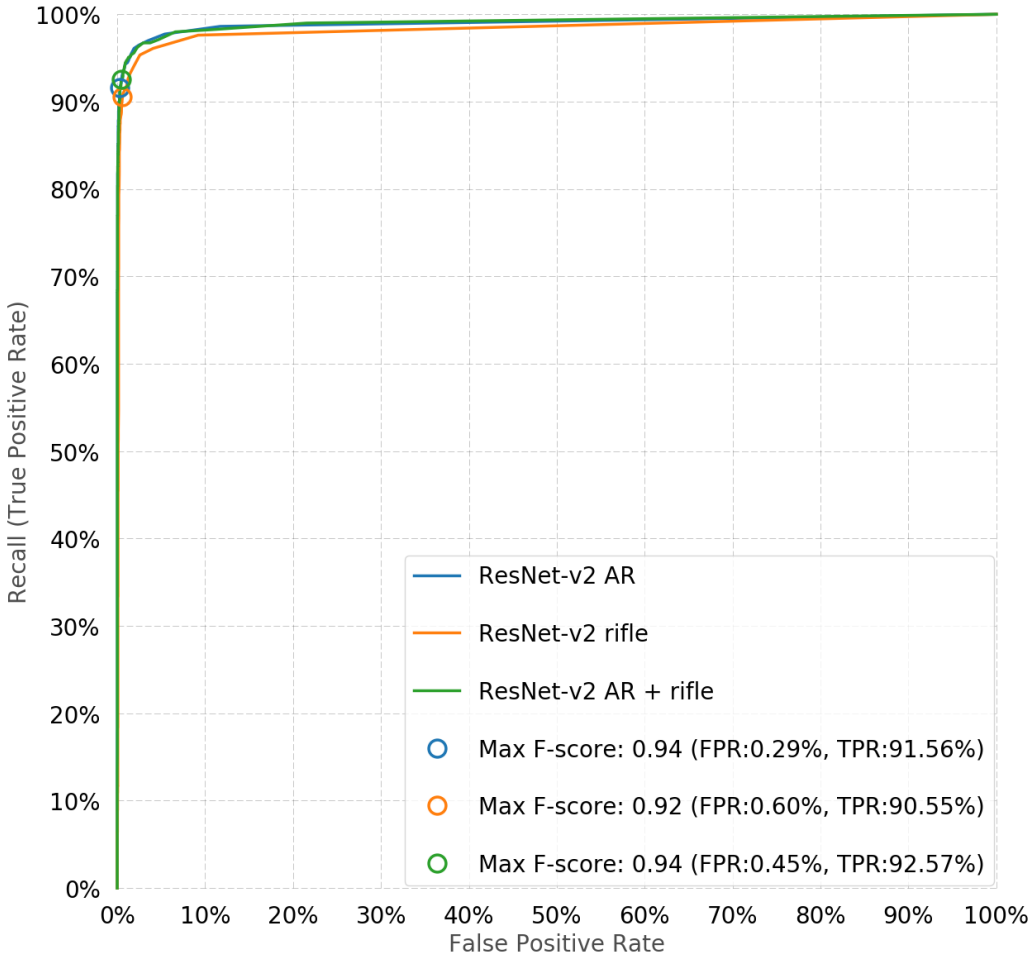


Figure 5.2. Inception-ResNet-v2 ROC curve for three category scores on all three test sets combined.

5.1.2 AK-47 Detector Performance on Jones' Test Images

The Inception models perform well as AK-47 detectors without any additional training and show significant performance increases from the first iteration to the latest Inception-ResNet-v2 model. Figure 5.3 provides the ROC curve for all five Inception models evaluated on Jones' [15] test set images; in the Appendix, Figure A.6 shows a closer view of the same curves. At the best F-score threshold, from v1 to ResNet-v2, the recall rate increased by eight percentage points while the FPR dropped by a factor of three. At its optimal threshold, the Inception-ResNet-v2 correctly identified 645 of 687 AK-47 images while only incorrectly identifying 10 of 7,045 negative images as containing an AK-47.

For the reasons discussed in Section 4.1.5, we cannot directly compare the performance of the Inception models to the algorithms implemented by Jones [15]. In order to avoid misleading the reader we do not provide Jones' [15] ROC curve in this section for a side-by-side comparison. Still, we can observe the recall when the FPR equals zero to obtain a sense of how strongly the Inception-ResNet-v2 model outperforms Jones' [15] algorithms. Instead of using the best F-score threshold for our classifier, we instead use the highest AR category score that generated a false positive on a negative example. Classifying any image with a score above this threshold as an AK-47, we obtain a FPR of zero and a recall of 86.2%. At this new threshold, our algorithm still classifies 592 out of 687 AK-47 examples correctly without generating any false positives. Observing the left side of the ROC curves from Figure 2.2, all of the algorithms begin to generate false positives before producing recall rates above 78%. This simple comparison removes our largest difference in calculations, the FPR rate, and inspires confidence that the Inception models improve classification error rates in their out-of-the-box configuration. See Appendix A.1.2 for a closer view of Figure 5.3 and also a table of key performance metric values at the best F-score threshold for each model.

5.1.3 AK-47 Detector Performance on Internet Images

Desiring to stretch the best performing Inception model further, we evaluate Inception-ResNet-v2 on the *AK-47 Internet Images* and *Rifle-Like Images* test sets described in Section 4.1.1. Figure 5.4 portrays the ROC curves for the Inception-ResNet-v2 model on these two new test sets with the curve produced previously for Jones' [15] test set. Not surprisingly, performance of the model drops when we test it against more naturally occurring images of AK-47s versus the AK-47 video stills. On the *AK-47 Internet Images* test set, at its best threshold the Inception-ResNet-v2 model classifies 98 out of 107 of the images containing AK-47s correctly, and only produces two false positives out of 143 negative examples. Even when attempting to fool the model by providing images of rifle-like objects, the Inception-ResNet-v2 model performs reasonably well; it only classifies 18 of 100 of the images incorrectly. As before, Appendix A.1.3 contains more detailed metrics for these tests.

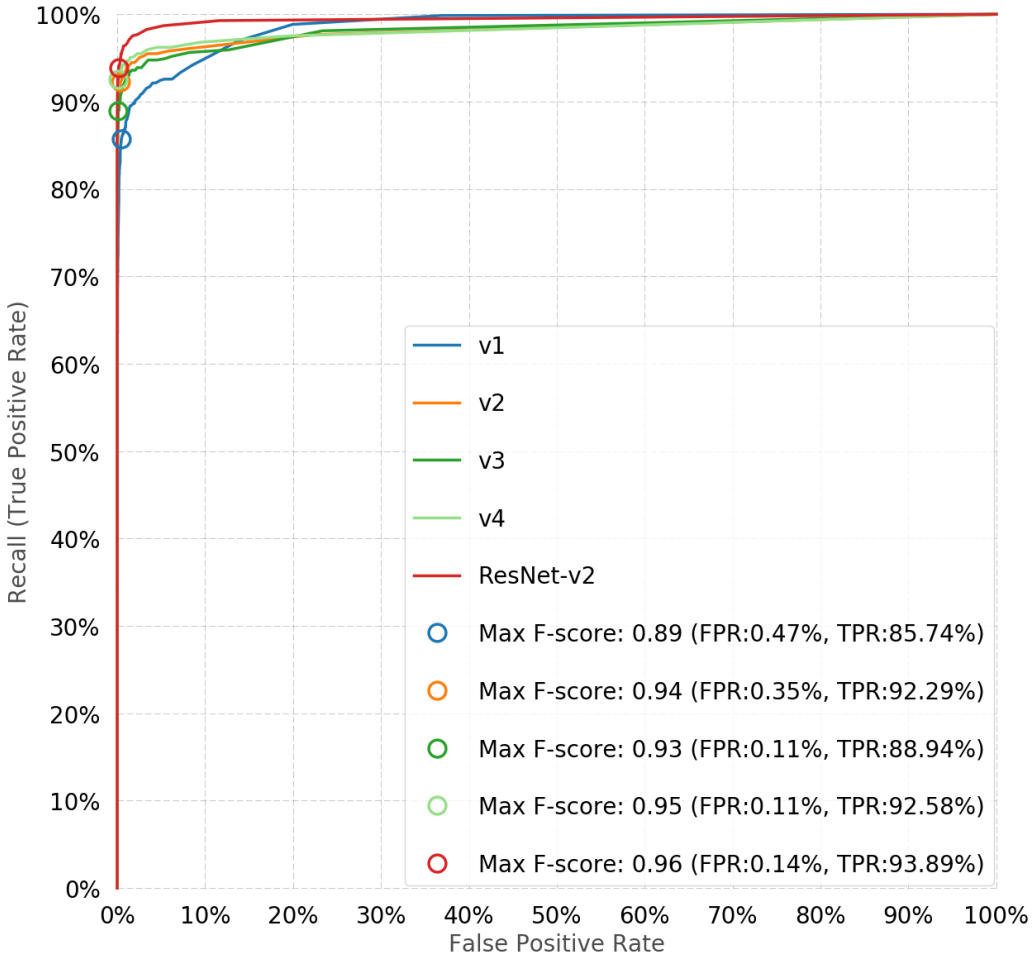


Figure 5.3. AK-47 detector performance using the AR category for all Inception models on Jones' [15] test set.

5.1.4 AK-47 Detector Missteps Analysis

An analysis of the Inception-ResNet-v2 model's missteps provides insight into the algorithms performance and the kinds of mistakes it makes. Interesting trends exist in these missteps in terms of both false negatives and false positives. Starting with false negatives, we first explore failures in Jones' [15] test set due to the opportunity to compare stills from the same video that the algorithm classified correctly to those it missed. In about 30 of the false negatives, the algorithm missed when an object of a known category in the background became the central focus of an image. For example, a video sequence of a man shooting an AK-47 in front of a mobile home produced 24 of the false negatives. The algorithm missed when the camera zoomed out and included the entire mobile home in the center of

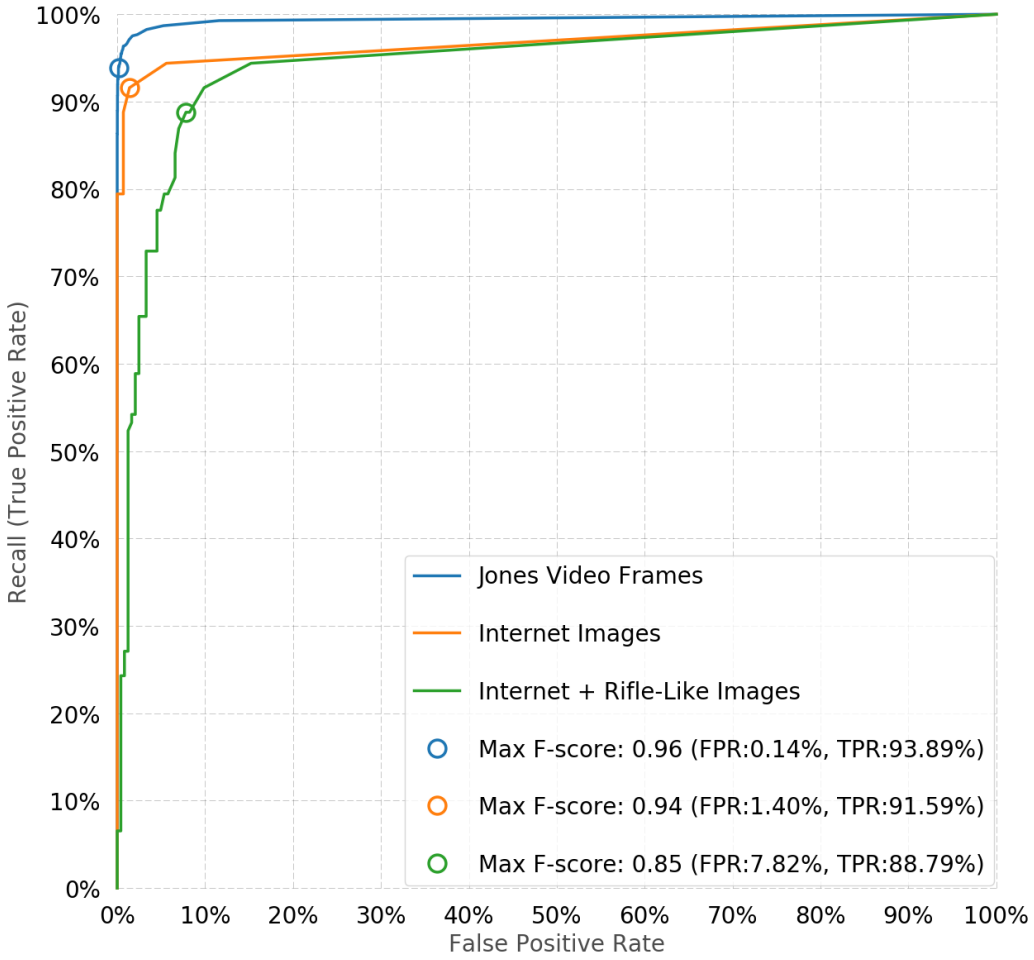


Figure 5.4. Best AK-47 detector performance using the AR category on internet test sets.

the image behind the shooter. Table 5.1 presents the top-three classification scores for a distant-view image, which Inception-ResNet-v2 misclassified at our best F-score threshold, and the scores for the close-up view of the AK-47 which the model classified correctly. In the distant view, the algorithm detected the mobile home so strongly that it did not produce a significant score in any other category.

Table 5.1. AK-47 detector top-three category score comparison for false negatives versus true positives in same video sequences.

Image	Cat. 1	Score	Cat. 2	Score	Cat. 3	Score
Mob. Home(-)	mobile home	0.946	revolver	0.002	boathouse	0.001
Mob. Home(+)	assault rifle	0.586	mobile home	0.159	rifle	0.158
Lawn(-)	giant schnauzer	0.121	Appenzeller	0.080	lawn mower	0.064
Lawn(+)	assault rifle	0.755	chainsaw	0.086	rifle	0.065
Boots(-)	cowboy boot	0.121	plow	0.080	bow	0.064
Boots(+)	assault rifle	0.070	valley	0.060	worm fence	0.046

This table compares a false negative (–) and true positive (+) frame from the same videos of Jones' [15] test set to describe causes of error. We include only the first category description instead of the full name of each ILSVRC category.

In a second instance of failure, pictured in Figure 5.5, a woman fires an AK-47 on a bright green lawn. The Inception model classifies all but the last two frames of this sequence correctly. When the woman moves the weapon from the firing position to below her hips and turns towards the camera it appears that the lawn in the background strongly influences the classification scores. Table 5.1 shows that the classifier produces its strongest scores for two dog breeds and a lawn mower for the last two frames. Examining the ILSVRC 2012 training set for the dog categories, we find that a significant portion of the images are of the dogs sitting on bright green lawns. The third highest classification score of lawn mower also supports this hypothesis. While the algorithm performed well enough on the ILSVRC 2014 challenge for us to know that it is not simply relying on bright green backgrounds to detect these categories, it is interesting to note that the presence of grass does significantly influence the category scores.

(a) True positive: Lawn (+). (b) False negative: Lawn (–).

Figure 5.5. Stills from Jones' [15] test set of an AK-47 video from YouTube. See Table 5.1 for these stills' top three category scores. Adapted from [60].

A final phenomenon for discussion that negatively affected AK-47 detection were stills in which a camera lost focus and an image became pixelated around the AK-47, perhaps due to rapid motion of the weapon during firing. In the third example pair in Table 5.2, this loss of focus led our algorithm to pick up the image's more clearly defined objects, like the boots on a man in the background of the image. These three examples cover a majority of the false negatives produced by the Inception model and develop some intuitions for the causes of failure.

Let us shift focus to false positives produced by Inception-ResNet-v2. There are fewer on Jones' [15] test set, but one trend exists, and the internet and rifle-like images produce the most interesting trends for discussion. For Jones' [15] test set, 5 of 10 false positives contained military members in uniform without any weapons. Table 5.2 records the top classification scores side-by-side with scores for the assault rifle category for these five images. This trend is not surprising: many of the training images for the assault rifle category contain military members in uniform. Thus, from the model's perspective a military uniform in a picture increases the likelihood that an assault rifle also exists in the image. Since assault rifle earned the top category score for only one of these five images, and the remaining four images' scores are close to the best F-score threshold, this trend does not produce significant concern in light of the Inception model classifying other images

with military members in uniform correctly. We could probably correct both this bias and the green lawn bias by including more training examples to counter the biases; they do not represent a significant algorithm flaw. Moving on to the trends present on the internet and rifle-like image sets, 6 of 19 false positives on these sets came from images containing paintball guns. Considering the similar features between assault rifles and paintball guns, and the fact that model does not possess a paintball gun category, this seems like a reasonable mistake. Nine other false positives came from images of men holding large wrenches, power drills, a pitch fork, and a fishing pole in carry positions common when holding a weapon, like leaning on a shoulder, 45 degrees across the body, or underarm. Again potential feature similarities make these errors seem reasonable. Another false positive was for an image containing the undercarriage of an automobile. The algorithm placed assault rifle as the eighth highest category for the image with a score of 0.022, well above the best F-score threshold for the rifle-like images set. Perhaps the edges and color patterns in the pipes and struts under the car resemble those features the algorithm learned from images of assault rifles. The remaining three false positives are harder to reason about and justify. They contain images of groups of people with no obvious patterns that could confuse the algorithm. Such mistakes, those that a human does not make or struggles to understand, are a weakness of algorithms incorporating ANNs which we will discuss more thoroughly in Chapter 6.

Table 5.2. Top category and assault rifle scores for AK-47 detector false positives on military members in uniform.

Image	Top Category	Score	AR Rank	AR Score
1	mountain bike	0.197	5	0.042
2	military uniform	0.300	4	0.046
3	assault rifle	0.265	1	0.265
4	bulletproof vest	0.101	3	0.072
5	stretcher	0.188	4	0.055

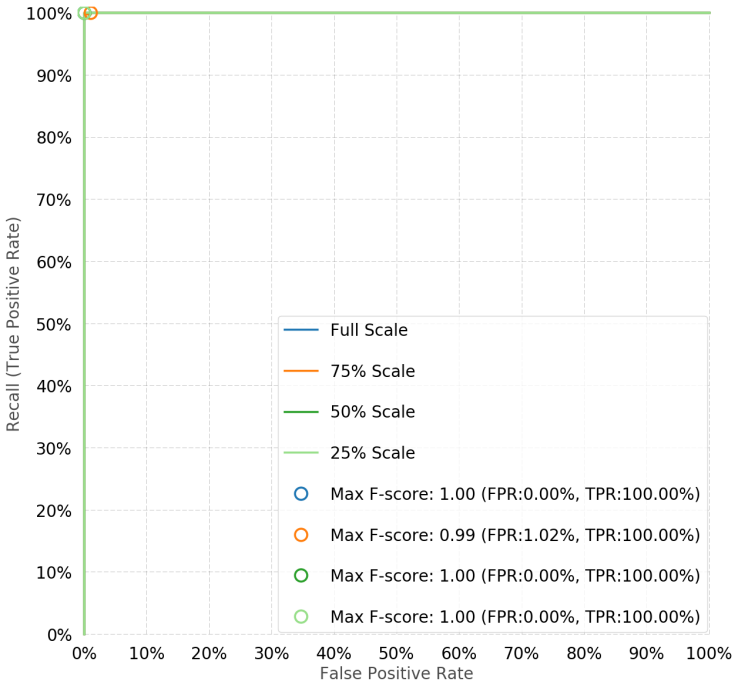
This table provides the top category scores for five false positive images from Jones' [15] test set on images that contained military members in uniform. Any score for the AR category above a threshold of 0.036 produced a false positive. The first image was of a Marine in camouflage dancing next to two children on bicycles. The remaining four images were from the same video sequences of army soldiers running a relay race in camouflage in front of a building. The table includes the rank, with one being the highest, that the assault rifle category score received out of all 1,000 possible classifications. We include only the first category description instead of the full name of each ILSVRC category.

5.2 Retrained Inception-v3 Ship Detector Performance

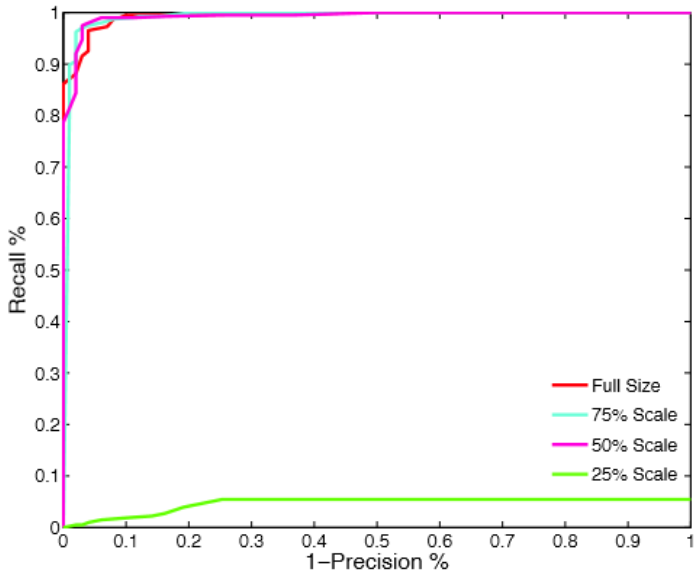
This section includes our results from applying transfer learning techniques and retraining the Inception-v3 model as discussed in detail in Section 4.2.4. It is important to remember that the models discussed in this section benefit from previous training on the 1,000 categories of images from ImageNet and that we only update the last fully connected layer. We present performance of the first retrained model using only Camp's [16] training images on his test set. Next, we provide the results for our second model iteration, which we retrained by adding the *Ship Internet Images* training set to Camp's [16] training set, on both Camp's [16] and the *Ship Internet Images* test sets, defined previously in Table 4.4. Finally, we explore missteps of the retrained Inception-v3 model by examining images it misclassified.

5.2.1 Ship Detector Performance on Camp's Test Images

The retrained Inception-v3 model achieves near-perfect performance when classifying Camp's [16] test images. Figure 5.6 depicts a comparison of Inception-v3 to Camp's [16] best performing model at the four largest scales. As shown in this figure, the Inception algorithm only misclassifies a single *no ship* image at the 75% scale. Moving to Figure 5.7, we provide a closer view of the retrained Inception-v3 model's performance at seven scales to compare to Camp's [16] best performing model across all scales. Observing the Inception-v3's performance at the smallest scale, we see that it outperforms Camp's [16] HYBRID model at full scale. The Inception-v3 model achieves strong scale invariance as a ship detector; a desirable trait for the UAS use case developed in Section 1.4. See Appendix A.2 for an ROC curve depicting all eight scales and a table of performance metrics at the best F-score threshold for each scale.

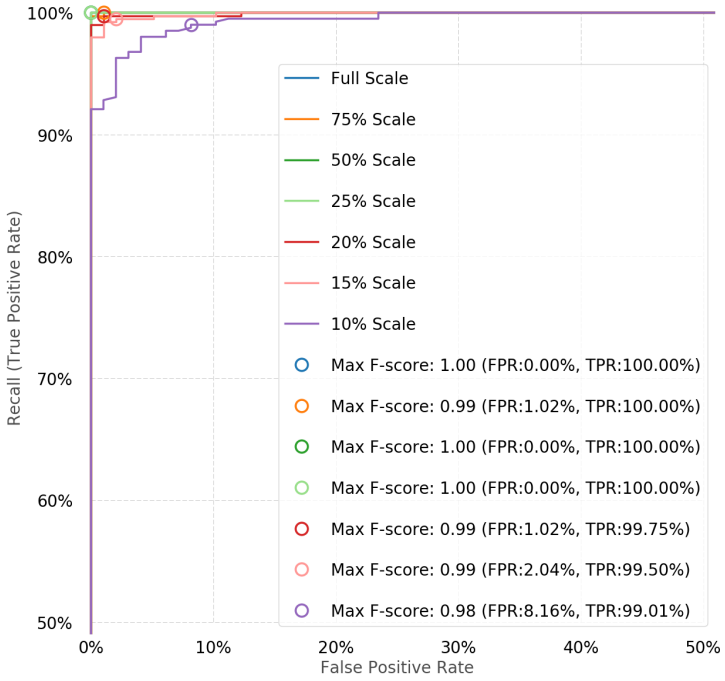


(a) Inception-v3 ROC Curve.

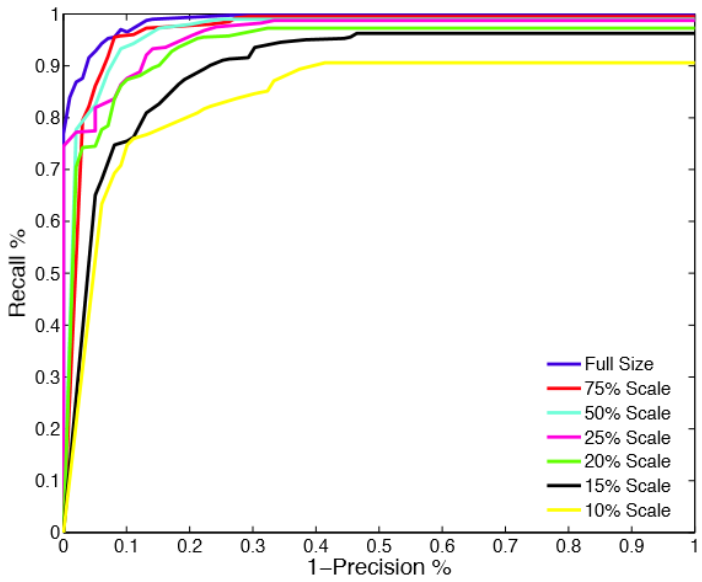


(b) Camp's DPM ROC Curve.

Figure 5.6. Inception-v3 ship detector performance when retrained on Camp's [16] training set compared to his DPM detector. Camp's [16] DPM detector achieved the highest performance of all his algorithms at scales above 50%. Due to poor performance at smaller scales Camp [16] only presents down to 25% scale in this plot. Source(b): [16, Figure 18].



(a) Inception-v3 ROC curve close-up.



(b) Camp's HYBRID ROC curve.

Figure 5.7. Inception-v3 ship detector performance when retrained on Camp's [16] training set compared to HYBRID detector. Camp's [16] HYBRID detector achieved the best performance across all image scales. We provide a close-up view of the ROC depicted in Figure 5.6 including smaller scales to match Camp's [16] plot. Source(b): [16, Figure 18].

Because he separates the ROC curves by scale, Camp [16] does not test his algorithms' ability to use a single threshold and classify images across multiple scales. This capability would add utility to the ship detector we describe in our UAS scenario from Section 1.4. Thus, in Figure 5.8 we combine images from full scale down to 25% scale in order to measure the retrained Inception model's classification performance across the scales. Examining the *Camp Training Only* curve, we see that Inception-v3 achieves scale invariance and still retains near perfect performance when performing classification using the single best threshold of 0.450 for multiple scales. The algorithm generates a single false positive and false negative across four different scales of each of the 405 positive examples and 99 negative examples. Still, identifying a small-scale ship when it is the focus of the image remains an easier problem than having to identify the ship in an image covering a long horizon; this harder problem is the focus of our next tests.

5.2.2 Ship Detector Performance on Internet Images

This section describes the performance of the *Added Images* model produced by a second iteration of retraining that incorporated training and test images from the more challenging *Ship Internet Images* set. We evaluate the *Added Images* model (described in Section 4.2.4) that achieves the highest validation set accuracy during training while only completing 1,000 training steps. This model had additional *no ship* and *ship* examples available as described in Section 4.2.1. Figure 5.8 displays a comparison of the performance of the first iteration of Inception-v3, the *Camp Training Only* model, to our *Added Images* model on Camp's [16] test set across four scales. The *Added Images* model performs worse but not significantly; it still achieves an F-score of 0.99. The *Added Images* model does generate 20 false positives out of 396 negative examples; however, 14 of these errors are repeats of the same images at different scales. Moreover, it only generates two false negatives across all four scales of ship examples.

Confident that our *Added Images* model still performs well on Camp's [16] images, we provide a depiction of the algorithm's stronger resistance to false positives on the *Ship Internet Images* test set in Figure 5.9. While at some thresholds the *Camp Training Only* model performs slightly better than a random guess, its best F-score still comes from classifying all 120 difficult test images as ships. Comparing the algorithms at their best F-score thresholds, the *Added Images* model increases its F-score over the *Camp Training*

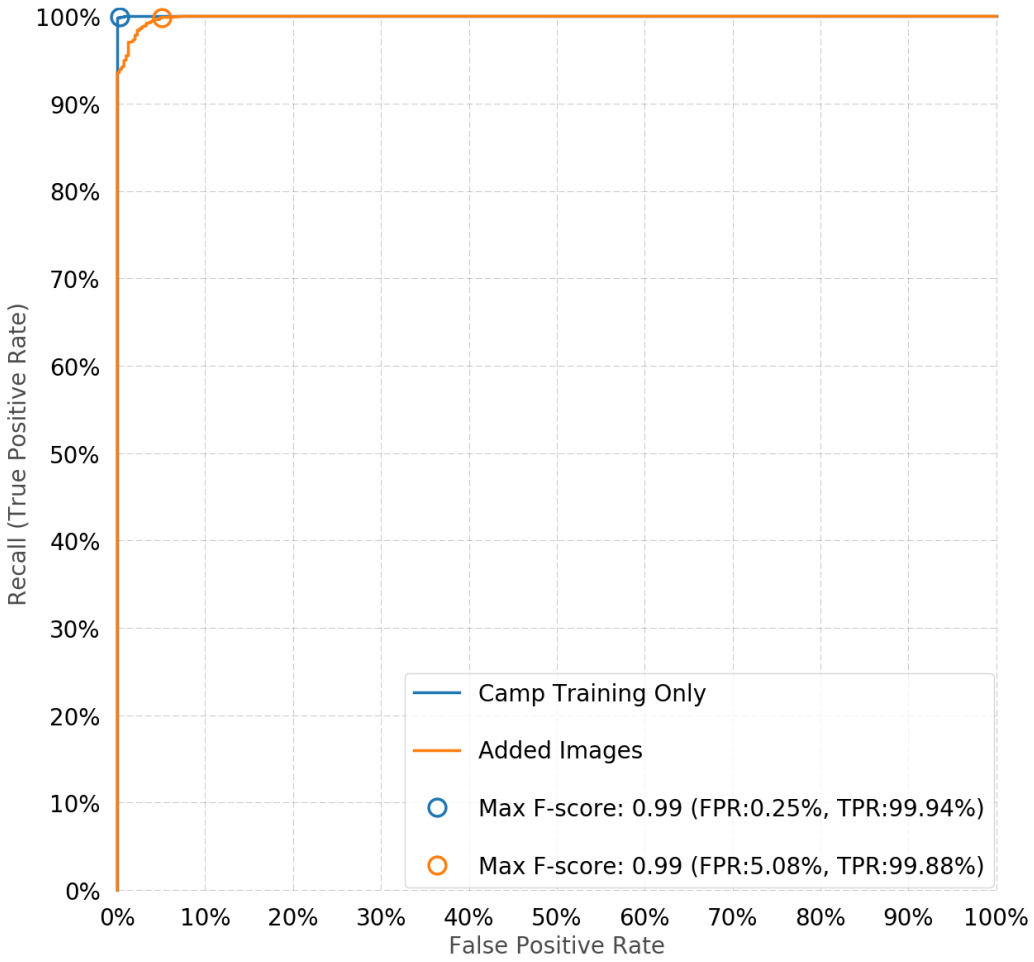


Figure 5.8. Inception-v3 iterations compared on Camp's [16] test set from full to 25% scale combined. This figure shows performance of the first iteration Inception-v3 model, retrained only on Camp's [16] images, and the second iteration, retrained with the added images discussed in Section 4.2.1. These curves show performance of the models across Camp's [16] test images combining the following scales: full, 75%, 50%, and 25%. We discuss the second iteration model performance in Section 5.2.2.

Only model by 15 percentage points. At its best F-score threshold, the *Added Images* model manages to classify 53 of 60 *ship* examples correctly while generating 24 false positives out of 60 negative examples. The *Added Images* model does not perform well enough to implement as is; yet, our results support that with more relevant training examples Inception-v3 can learn to identify ships on the horizon and distinguish large ocean debris from ships.

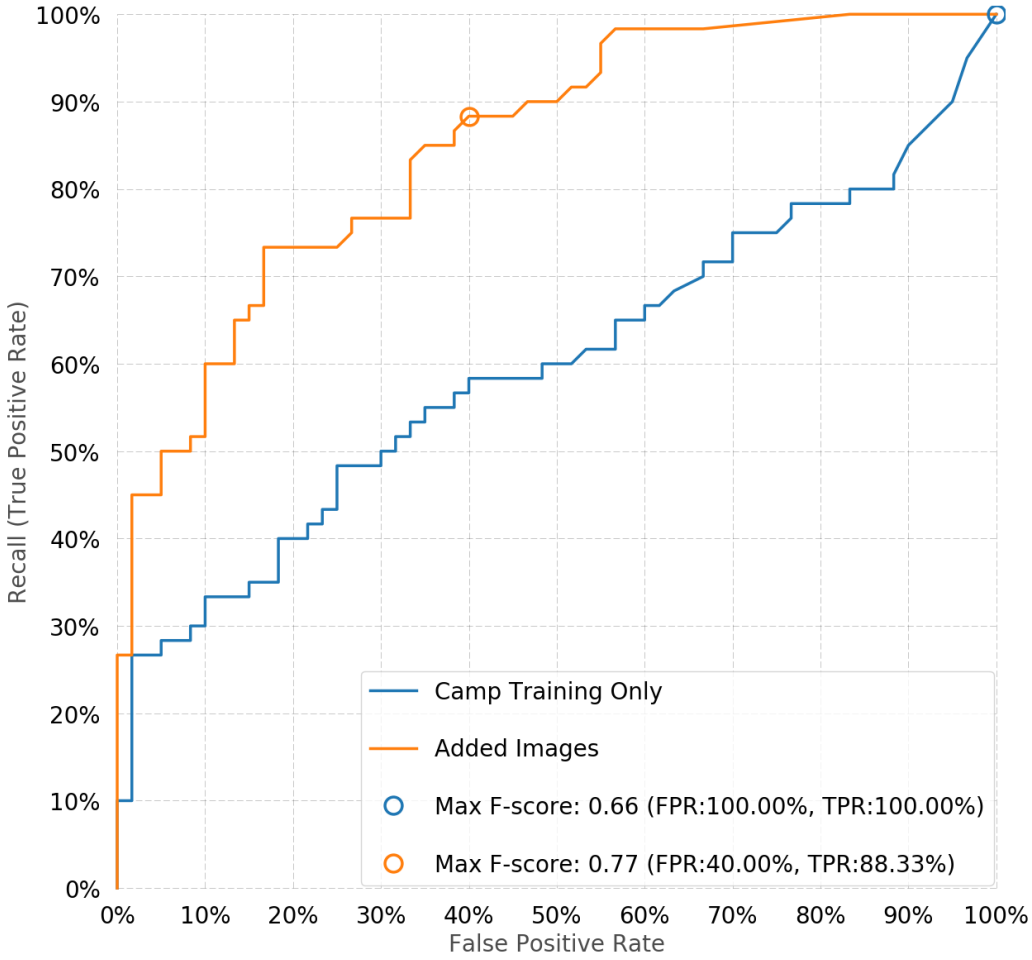


Figure 5.9. Inception-v3 iterations compared on the Ship Internet Images test set.

5.2.3 Ship Detector Missteps Analysis

This section discusses the errors for both iterations of the ship detector on Camp's [16] test set and the *Ship Internet Images* set. We start with the first Inception-v3 iteration (the *Camp Training Only* model) evaluated across four scales. At its best F-score threshold the model misses two images at scales above 20%. It classifies a full-scale image of a large ice shelf on a hazy horizon as a ship. It is easy to imagine the algorithm identifying similar features to that of a ship's bow at the pointed edges of the ice, driving the ship category score to a value of 0.516 which fell above the 0.450 threshold. Looking at the single false negative, the *Camp Training Only* model classifies a 75% scale image of a cargo ship as not containing a ship. The cargo ship in this image sits in front of a coastline with a water

tower in the distance. Due to Camp's [16] training set containing several negative examples with coastal structures, the algorithm may have been overly influenced by the presence of the water tower. This ship image scored 0.446 for the ship category, only 0.004 below the threshold to classify it as a ship. We omit a discussion of algorithm misses below 25% scale as the misclassifications do not contain obvious trends.

Examining the *Added Images* model misses on the *Ship Internet Images* test set, we find two trends worth discussing. In terms of false positives, even with the additional land-based shipping containers in the training set, the algorithm still classifies eight images with shipping containers floating in the water as ships. It also classifies five box-like pieces of trash as ships; Figure 5.10 depicts two of these images. The distinct linear features of these images are similar to that of many of the training images for ships. All seven of the false negatives came from challenging images of ships distant on the horizon, again Figure 5.10 depicts two of these example images. Since we know the algorithm can detect ships at smaller scales, there are two options to improve performance on such examples. Either modify the algorithm to slide a window along the horizon, feeding less of the entire view through the algorithm at a time, or simply provide more positive training examples with ships at this distance and in different positions in the frame. These two trends suggest a need for more training examples; we did not spend the time to procure them as building the additional internet images set already went beyond the original scope of this thesis. Still, this discussion should serve as a starting point for the next steps in developing a training set for another iteration of this algorithm in pursuit of refining it for operational use.

(a) False positive. Source: [61].

(b) False negative. Source: [62].

(c) False positive. Source: [63].

(d) False negative. Source: [64].

Figure 5.10. Example error images from the Ship Internet Images test set.

5.3 Screenshot Detector Performance for CNNs Trained from Scratch

This section presents the performance of Oxford's VGGNet, Google's Inception-ResNet-v2, and Microsoft's ResNet-v2 (50 layer version) as screenshot detectors. Our best model achieves the same high-level of performance as Sharpe's [17] best combination of all four feature sets. Figure 5.11 depicts the ROC curves for both iterations of the three models. All of the trained models show promise as screenshot detectors but the Inception-ResNet-v2 model, trained for 50,000 steps, stands out as the top-performer. At its best F-score

threshold, this model achieves the highest recall by three percentage points over the next closest model by correctly identifying 2,314 of 2,420 *screenshot* examples. At the same time it produces the fewest false positives, classifying only 11 of 3,644 *other* images as screenshots. Sharpe [17] did not provide an ROC curve for her best performing combination of feature sets, but we reproduce the numbers from Table 2.4 to compare her algorithm that achieves the highest F-score to our best in Table 5.3. From this table we see that the algorithms obtain near-identical performance; Sharpe's [17] best algorithm surpasses the Inception-ResNet-v2 algorithm when including the third significant digit. As discussed in Section 4.3.3, our algorithm evaluated all 6,064 of Sharpe's [17] archived images while her numbers are averages from ten-fold cross validation across a selection of only 4,800 of the images. Also, mentioned in Section 2.2.3, Sharpe's [17] negative test images are grayscale. This test set trait did not provide an advantage to her algorithms as they did not extract features based on color. In contrast, the Inception-ResNet-v2 model is able to detect color patterns in images; however, since we train the Inception model using only color images for both negative and positive training examples, we do not gain advantage from this test set trait either. Rather, we hinder the Inception model as it does not benefit from any useful color patterns it learned during training and simply has to rely on other features, such as higher level combinations of edges and shapes, to classify Sharpe's [17] negative examples.

Table 5.3. Screenshot detector performance comparison for best F-score models. Adapted from [17, Table 4.9].

Algorithm	**Accuracy**	**Precision**	**Recall**	**F-score**
Sharpe's All Features	0.980	0.997	0.963	0.980
Inception-ResNet-v2 (50K)	0.981	0.995	0.956	0.975

5.3.1 Screenshot Detector Missteps Analysis

Once more looking at the algorithm misses, this time for the Inception-ResNet-v2 model at its best F-score threshold, we find three observations worthy of discussion in the false negatives and a single trend in the false positives. Figure 5.12 displays the images discussed in this section and Table 5.4 provides the images' category scores produced by the Inception model. The first observation comes from the only false negative in a series of screenshots

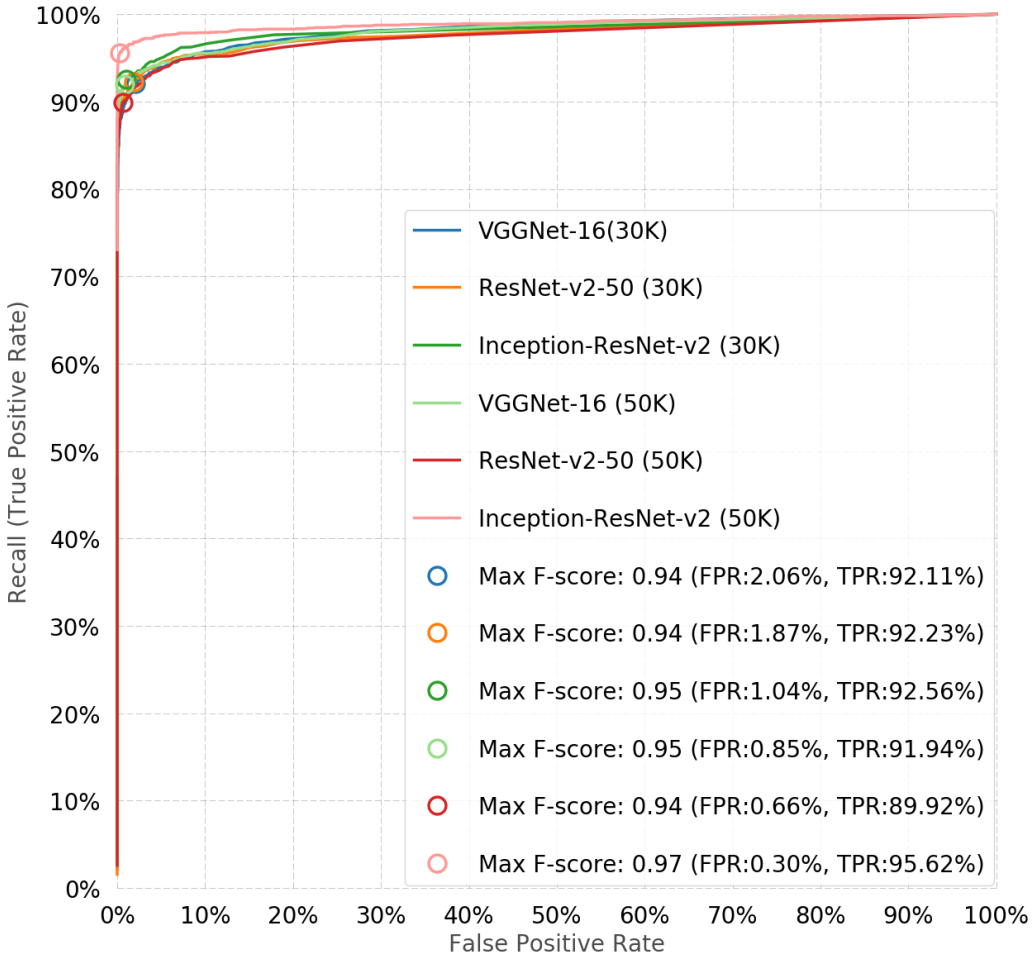


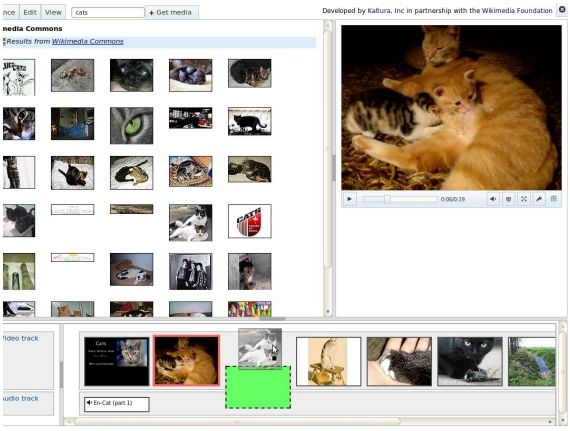
Figure 5.11. VGGNet, Inception-ResNet-v2, and ResNet models compared on Sharpe's [17] entire image set.

from a Linux terminal. Comparing the images in Figure 5.12a and Figure 5.12b, the sole difference is the presence of more text in the latter image. This extra text increased the algorithm's confidence that the second image contained a screenshot by 0.126; enough to break the threshold and allow a correct classification. This observation supports the hypothesis that the model learned to associate typed text as a feature of screenshots.

(a) False negative. Source: [65].

(b) True positive. Source: [66].

(c) False negative. Source: [67].



(d) True positive. Source: [68].

Figure 5.12. Example false negatives (left) and related true positives (right) from the Screenshots [17] test set.

Table 5.4. Category scores for images in Figure 5.12.

Image	Screenshot	Other
Terminal(-)	0.828	0.172
Terminal(+)	0.954	0.046
Real Image(-)	0.340	0.660
Real Image(+)	0.999	1.41E-12

This table provides the category scores from the best Inception-ResNet-v2 model for the images in Figure 5.12. We classify all images using a best threshold of 0.897. The examples are from similar images in the Screenshots test set to facilitate a discussion of what features the model relies on for classification.

A second trend comes from a pattern of misses for screenshots that include large depictions of real images. The best model missed 15 such images. Figure 5.12c depicts one of these false negatives and Figure 5.12d depicts a similar screenshot, with an even larger area containing real images, that the algorithm classified correctly with almost perfect confidence. It is interesting to note the uniform and perfectly white margins separating the real images in the latter image. Such features rarely occur in our *other* category examples whereas the earth-toned colors of the less-defined margins of the false negative occurred in a portion of the real images. It appears that the presence of the more obvious artificial margins allows the algorithm to detect that the real images in Figure 5.12d are a part of a screenshot. A final source of bias in the false negative examples is the tendency of the algorithm to classify screenshot examples consisting of mostly blue colors as *other* category images. The algorithm misclassified 21 blue-hued screenshots. Looking to the *other* category training set, the source of this bias becomes obvious. As discussed previously, we train the algorithm with categories from ImageNet including buildings, highways, coasts and more. A majority of these types of images contain large swaths of blue sky. In contrast, our training set for the *screenshot* category did not contain significant numbers of blue-hued screenshots; about five percent were mostly blue. For this reason, it appears the algorithm became overly responsive to the prevalence of blue pixels as a predictor for the *other* category. One potential fix for this bias is to follow Sharpe's [17] lead and forgo considering color features. We could do this by converting all training and test images to grayscale during

image preprocessing for the model; however, we did not pursue correcting this bias and suspect limiting the model in this manner may have other undesired performance effects.

Of the 11 false positives, ten contained furniture, houses, buildings or man-made structures. The high prevalence of linear features, large patches of consistent pixel intensity, and manufactured patterns in these images proved challenging for the algorithm as these feature prevail in screenshots. Even with these misses, the algorithm proved highly resistant to false positives in other images where the same features exist, but are somewhat less prevalent.

5.4 TensorFlow Models for DOD Operating Environments

Beyond classification performance, our experiments assess feasibility in terms of runtime for implementing TensorFlow-based object detectors in simulated DOD use cases. We first compare the training and evaluation time of our algorithms to the times from the three previous theses captured in Chapter 2. Then we apply our image classification runtimes to three potential DOD applications to develop an intuition for the prospective time savings available through our TensorFlow-based object detectors compared to manual analysis.

In general, if run on the same hardware, the deep CNNs we implement in TensorFlow take longer to train and evaluate than preselected feature extractors of previous models. For example, we did not attempt to train the deep CNNs from scratch using only CPUs due to the expected week-long-training times. Still, the effort savings afforded to algorithm designers, who no longer have to manually adjust methods for feature extraction for every new object detector, make the extra-computational costs acceptable. Also, Table 4.6 proves that methods like transfer learning can negate the extra training time required, even in resource-constrained environments. When a classification problem warrants training a deep CNN from scratch, Table 4.9 illustrates a modern GPU and CPU combination allows our algorithms to complete training in 12 hours or less; a reasonable training cycle for testing many iterations of the algorithms. This is important as the algorithms often require hundreds of iterations of training in order to conduct a proper hyperparameter search to achieve optimal performance. With the largest computational costs levied during training, Table 4.3 shows that the slowest TensorFlow-based detectors we tested still classify images in less than one second in resource-constrained environments. Even though this evaluation runtime proves slower than previous methods, we believe the algorithms are fast enough to

remain relevant considering their improved accuracy and potential time savings over human-only analysis of images. Importantly, when leveraging current hardware and optimized data pipelines to feed images into the algorithms, deep CNNs prove capable of offering near-real-time evaluation of images. By achieving classification speeds of over 60 frames per second in such cases, deep CNNs show potential for analysis of live video feeds.

In order to provide a concrete example of how TensorFlow-based algorithms may perform in real-world applications, Table 5.5 adapts the training and classification times from Chapter 4 to predict the TensorFlow-based algorithms' runtimes in realistic scenarios. These time estimates provide a rough measure of speed of classification of images in three potential DOD applications. From our experiments we believe the DOD can benefit from TensorFlow and deep CNNs in deployed, unmanned, and enterprise level systems and environments.

Table 5.5. Adaption of object detector runtimes in potential DOD environments.

Model	**Hardware**	**Batch Size**	**Images**	**Time**
Deployed				
Inception-v3	Computer(CPU)	1	5,000	31m
Inception-ResNet-v2	Computer(CPU & GPU)	100	10,000	3m
Unmanned				
Inception-v3	UAS(CPU)	1	3 stills	1.25s
	UAS(CPU & GPU)		6 stills	0.25s
Enterprise				
Inception-ResNet-v2	Data Center with	100	350M	14h
ResNet-v2 (50 Layers)	100x (CPU & GPU)			7h

We derive these time *estimates* by dividing the number of images by the images per second values in Tables 4.3 for rows listed with a *batch size* of one and Table 4.10 for rows with a batch size of 100. This table is meant to provide a simple estimate to guide the reader to the right order of magnitude for the time savings offered by employing the listed algorithms through TensorFlow on specific hardware combinations. The UAS example considers a system with three cameras or six cameras, providing 180 or 360 degree sensing respectively.

TensorFlow shows promise for application by deployed military units by easing time constraints for intelligence analysts at the battalion and squadron level and above. Employing only commonly available hardware, such as a laptop or desktop computer with a modern

CPU and at least 8GB of RAM, an analyst can now scan and tag images pulled from a confiscated hard drive in minutes instead of hours or days. If the analyst has access to a device containing a dedicated GPU our measurements show the time required decreases roughly by a factor of ten. Beyond the evaluation runtimes, referring back to the transfer learning times in Table 4.6, deployed analysts could also repurpose previously trained Inception models to identify images containing threat-specific objects, for example perhaps a local terror group's insignia. A typical laptop or desktop computer containing only a CPU can complete this retraining process in under thirty minutes. The resulting algorithm will analyze images with the same level of runtime efficiency and possibly, depending on the object and training images available, the same high-level of classification accuracy displayed by our retrained ship detector. With hardware found in deployed environments today, TensorFlow-based CNNs can benefit DOD users immediately.

TensorFlow-based algorithms' runtimes on devices with modest resources suggests utility for these algorithms in UAS. Recalling the ship-detecting UAS from Section 1.4, suppose that system had a port, starboard and forward-facing camera capturing live video. An UAS employing the Inception-v3 algorithm with a standard CPU could analyze stills taken from these three sources, scanning a video frame from each sensor every 1.25 seconds. A version of this same system with a GPU and three more cameras to allow 360 degree observation, could analyze a video frame from the six sensors four times a second. Once the algorithm detects an object of interest, it will alert its operator and send a segment of the saved video, including frames before and after the detection, back for human review. There are further time savings available if the six images, one from each sensor, are sent through the algorithm simultaneously as a batch. With TensorFlow-based CNNs, in the near future the DOD could achieve real-time detection of objects of interest with UAS.

A final area for discussion of time efficiencies is in employing TensorFlow-based algorithms at an enterprise level. For enterprise in the DOD, we consider a data center and the agency that employed Clara from Chapter 1. In terms of the number of images, we start with an estimated 3.25 billion images [69] shared on popular social media platforms a day. Assuming the intelligence agency would narrow down these images to specific sources and only collect about ten percent of them daily, we develop our hypothetical 325 million images. If the agency converted these images to TFRecords during collection and the data center could support 100 instances of a TensorFlow-based algorithm, we divide by the

images per second values from Table 4.10 and round up to the nearest hour to produce the estimate in Table 5.5. Although there are many more details to consider in optimizing the collection and storage process to facilitate efficient evaluation, it appears reasonable that our algorithms could flag images containing threat categories for human analyst review within hours of posting.

CHAPTER 6:
Conclusion

With the three experiments complete and our results presented in Chapter 5, a discussion of the answers to our research questions and the salient outcomes from the results remains. To begin, in light of the empirical results, we will answer our first four research questions from Section 1.6 and discuss why our results are relevant to the DOD. Next, we consider traits beyond just raw performance measures and seek to answer our fifth research question as to the high-level risks and benefits of employing an open-source machine learning library in the DOD. Last, to answer our sixth and final research question, we draw relevant next steps for the DOD from our work.

6.1 TensorFlow-Based Object Detectors' Performance and Why It Matters

In pursuit of our first four research questions we draw answers from the performance of the TensorFlow-Based object detectors summarized in the ROC curves and the time tables of Chapter 5. We start by comparing the training requirements of deep CNNs to previous methods in terms of compute resources and training data required, while also considering what our results reinforce about deep CNNs' versatility across datasets. We follow this discussion with another focusing on the tested algorithms' ability to scale for different operating environments in the DOD. Finally, we reflect on the improvement of classification performance and potential implications for future DOD systems incorporating these open-source solutions.

It is well known that Deep CNNs require more resources to train to convergence than legacy methods in terms of compute power and labeled data, yet their versatility across object detection problems makes them relevant to the DOD. We show this fact once again by considering the over one million training images from the ImageNet [1] dataset required to train the Inception models and the over 20,000 images we collected to train our screenshot detectors. In contrast, Jones' [15], Camp's [16], and Sharpe's [17] algorithms required only 6,806; 220; and 480 training images respectively. Still, our results affirm that the costs

associated with developing larger training sets and executing extra computations during training and evaluation prove worth paying. All three of the previous algorithm designers first studied the subjects of their detectors closely to understand the relevant features present in the objects they sought to detect. Then, they selected their feature extractors with care to match their specific classification problem. Next, they trained and tested many iterations of these feature extractors to optimize performance. Finally, they designed a data pipeline to run their images through the algorithms and produce a prediction. Deep CNNs do not demand such costly human involvement for each new use case. These algorithms instead incur a one-time design cost. Later, the algorithms only require sufficient training examples of the new object for detection and compute resources to repurpose the algorithm. For this reason, we were able to complete our experiments in the same amount of time each of the three previous algorithm designers had to construct a single object detector. With industry and academia paying these one-time-design costs for their open-source deep CNNs and the internet serving as growing source of training examples, the DOD can now, more than ever, afford to apply cutting-edge deep CNNs to meet a spectrum of its media analysis needs through open-source options.

Organizations of the Department of Defense demand software solutions for media analysis that function across a wide range of operating environments. Having discussed the steep training costs associated with TensorFlow-based object detectors, one might mistakenly assume that these algorithms only provide value to groups with significant compute resources at their disposal. Our results show otherwise. Conducting costly training up front, or repurposing trained networks through efficient methods like transfer learning, the DOD can deploy trained deep CNNs in more austere environments for evaluating images. By exploring the time required to classify images in both ideal and resource-constrained configurations, we prove that TensorFlow-based object detectors can operate on currently available hardware across many DOD operating environments.

The final and most significant conclusion from our empirical results lies in the classification performance improvements available through TensorFlow-based object detectors. Without any investment, in terms of development time or compute resources for training, TensorFlow's latest Inception model offers increased performance over previous AK-47 detectors and also shows promise in helping the DOD identify objects in images across hundreds of other categories. Further, historical improvement across five iterations of the Inception

model suggests that Google will continue to update and release future versions incorporating the latest advances in the field. Looking at repurposing an Inception model for a new task with transfer learning, we found the model to achieve near perfect results on Camp's [16] test set; increasing recall rates by 10 to 20 percent across image scales while reducing false positive rates. With this performance increase, we also prove that deep CNNs persist in their applicability for DOD media analysis problems even when limited numbers of training examples exist. Finally, in retraining a deep CNN from scratch, we achieved nearly identical results to Sharpe [17] on her test set. While this result is not a performance improvement, her task lends itself to human-selected feature extractors. Features applicable for identifying screenshots, like strong linear features, continuous patches of pixel values, and text, are more obvious to a human eye than those required to distinguish more complex subjects in natural photos. This trait of screenshots facilitated Sharpe's [17] near perfect classification performance with an F-score of 0.980. Coming in with a slightly lower F-score of 0.975, the best deep CNN's ability to learn applicable features without direct human input remains equally impressive. Achieving a strong level of classification performance on this task, we illustrate that when possessing sufficient images the DOD can apply proven algorithms to new problems by training deep CNNs from scratch. In summary, by attaining F-scores above 0.95 on all three previous test sets, we demonstrate the ability of open-source deep learning solutions to assist members of the DOD across a spectrum of media analysis needs.

We believe these results matter to the Department of Defense. The three previous experiments and ours span only seven years. In that time, machine learning algorithms have matured from hand-crafted models for single-uses to the deep CNNs that successfully generalize across many unique image classification problems. These algorithms' classification error rates are low enough to start implementing them in systems to facilitate human-machine teaming, where machines ease an analyst like Clara's workload and increase her effectiveness. The openness and collaboration of academia and industry continues to spur progress more quickly than individual groups could achieve. Google's Inception stands as direct proof of this statement. Starting with the work of others, the company developed its own state-of-the-art model, entering it into the ILSVRC 2014 competition. Then inventing and sharing new methods, Google released three further iterations of the model. Finally, seeing an admirable idea from another industry leader, Microsoft, Google incorporated residual layers into the Inception model and attained its most accurate model to date, which

it also made public. Referring back to Figure 5.3, one sees that in our application of Inception as an AK-47 detector, this open collaboration reduced error rates by 50 percent on Jones' [15] test set between the first and last iteration. The DOD must find a way to include itself in this collaboration and bring to bear open-source progress in order to keep up with the current rate of change in the field of machine learning.

6.2 Benefits and Risks of Employing TensorFlow in the DOD

Working with multiple releases of TensorFlow for over a yearlong period, we strive to summarize our experience through an analysis of high-level risks and benefits of applying TensorFlow and deep CNNs in DOD work environments and defense systems. In setting out to accomplish this task, we also seek the answer to our fifth research question.

Beyond providing access to powerful machine learning models discussed in Section 6.1, the benefits of TensorFlow include abstractions through the API that facilitate collaboration in solving machine learning tasks, continual updates and improvements to the software, and scalability across compute platforms. Because TensorFlow provides low-to-high levels of programming abstractions through its sub-libraries, like TensorFlow-Slim, users of the software can quickly apply proven algorithms to new problems with the high-level abstractions or develop new algorithms with the low-level abstractions. These abstractions provide a framework for defining and discussing implementations of machine learning methods to facilitate collaboration in solving complex problems. A second benefit of the software is its pace of improvement. Where the DOD struggles to maintain its software libraries, TensorFlow stands out for rapid releases of updates. These include new functionality, iterations of algorithms with increased performance, TensorFlow implementations of the latest machine learning models from academia, updates to data pipelines and processes in pursuit of computational efficiencies, and more. Further benefits for the DOD are TensorFlow's cross-platform availability and the scalability of the software. In our tests we employed TensorFlow on Linux Ubuntu, Windows 10, and CentOS operating systems and found our scripts and experiments to work well across the platforms. Officially, TensorFlow supports Ubuntu, Mac OS X, and Windows; but, as the source code is public, it is possible to compile TensorFlow for many more systems. TensorFlow community members already compile source code and share necessary files for installing TensorFlow on systems like

the Raspberry Pi. This leads us to the final benefit for discussion, scalability. With several choices of open-source machine learning software frameworks, competition thrives among maintainers to make the most efficient and versatile implementations of proven algorithms and methods. The resulting efficiency and versatility of TensorFlow allows it to run on platforms as small as the Raspberry Pi while also scaling to run in production environments, like Google requires, through the TensorFlow Serving library. Also, Google offers cloud services with TensorFlow [70] running on hardware specialized for machine learning which could benefit larger organizations within the DOD. Such scalability encourages application of the software throughout all levels of the DOD. This limited discussion of the benefits of TensorFlow, coupled with the performance improvements discussed previously, should sufficiently motivate efforts to incorporate it in the DOD.

Adopting TensorFlow in the Department of Defense does involve some risks and challenges. We introduce these here and discuss steps for overcoming them in Section 6.3. First, the software library requires an understanding of computer programming that the average-candidate users for the software in the DOD do not currently possess. Even with access to online tutorials and the prerequisite-programming skill, understanding TensorFlow's data pipelines, queuing systems, and other processes requires significant study and effort. This proved especially true when attempting to import and train proven models with a new dataset. Where we discussed the library's rapid updates as a benefit, a consequence of such innovation is that the software provides multiple options to conduct the same task. This leads to confusion as to which sub-library and which function calls to employ when developing code. These multiple options do serve two purposes: they allow users to complete tasks at different levels of abstraction or preserve functionality as legacy methods and procedures get updated, marked for removal, and eventually discontinued. Still, multiple options can become a source of confusion to users learning how to employ TensorFlow. A final challenge the DOD will face in using TensorFlow stems from its permissive Apache 2.0 License. The license allows anyone to modify the source code of TensorFlow and release it under a new license. Acquisition professionals seeking to incorporate TensorFlow into a DOD program will have to ensure that contractors do not make modifications to the software library, copyright the slightly modified version, and sell it to the DOD as a proprietary software library. These risks and challenges are not inconsequential; regardless, we believe the DOD

can accept and overcome them to reap the benefits available through open-source software like TensorFlow.

A final realization necessary to responsibly apply deep CNNs in the DOD requires recognizing and appreciating the errors such algorithms make. As we showed in an analysis of each object detectors missteps in Chapter 5, deep CNNs do not make the same types of mistakes humans make. They are capable of missing objects obvious to a human analyst and also prove able to generate false positives in cases that would not deceive the human eye. Appreciating this fact narrows the types of problems with which these algorithms can assist. Specifically, current deep CNNs do well filtering and prioritizing information for human review, serving on a human-machine team, vice making independent decisions which bear significant consequence, like automatic weapons release. A benefit of initial human-machine teaming is the ability of a human to identify patterns in an algorithms' mistakes, especially for false positives, and add misclassified examples to a training database with the goal of improving the algorithm's performance. For example, with a deep CNN AK-47 detector, a program would first sort images from a confiscated hard drive from the highest to lowest score of containing a weapon. The human reviewer could quickly inspect the highest-ranking images for actionable intelligence. When time permitted review of the remaining images, the analyst could complete her inspection and add any identified false negatives to the algorithm's training set. In this way it is possible to improve the AK-47 detector in the future without requiring a complete redesign of the model. In applying deep CNNs to problems that facilitate such human-machine teaming, the DOD can mitigate risk and gain an appreciation for their strengths and weaknesses while considering their potential for future independent applications.

6.3 Next Steps for the DOD to Employ Open-Source Machine Learning Solutions

In an effort to address some of the risks of using TensorFlow, and answer our final research question, this section explores potential next steps for DOD organizations hoping to benefit from open-source machine learning solutions. Our research confirms that TensorFlow offers a needed capability to the Department of Defense, a first step in a more complex problem of providing access to similar software for groups across the organization. Significant work remains to implement the technology and software we explored in this thesis for operational

use cases. A potential next step is to compare available libraries and select one or more for use on DOD systems and networks. Narrowing the selection to a few of the most promising frameworks mitigates risk of employing a sole framework, inspires further competition among open-source communities, and still encourages collaboration as DOD groups can share solutions they implement within the same open-source machine learning libraries. A second step is to make it known that the selected software is available to DOD users. While we showed that policy exists to allow open-source products in the DOD, the organization must inform personnel of their ability to employ the software and provide specific guidance about licensing considerations and terms of use. Further, information assurance policies must permit installation of the software on government owned systems; this may require certification of the software by DOD security professionals.

Once given access to the software, personnel will require training. Thus, DOD organizations must develop a training pipeline to provide the prerequisite knowledge of a programming language, a basic understanding of the principles of machine learning, and introductory training on application specific algorithms, like deep CNNs, to gain an appreciation for their strengths and the types of mistakes they make. Fortunately, the required training topics are available through online courses and material [32], [71], [72] and the DOD could train personnel efficiently by developing certifications for specific personnel who complete a combination of online courses. These three steps can move the DOD closer to delivering the ability to apply open-source object detectors in media analysis tasks to intelligence analysts at many levels of the organization.

As a potential strategy to reduce training requirements, the DOD could also contract a company to build and maintain a graphical user interface that makes calls to an open-source machine learning library. This project would require minimal software development but would allow everyday users access to the deep learning algorithms, like the Inception models, without requiring significant instruction. The interface could also allow users to apply transfer learning to the Inception models by supplying directories containing example images for new categories. In only contracting the company to provide the graphical user interface, the contractor could not make the underlying library proprietary and the cost of employing TensorFlow would remain low. Completing this step, everyday members of the DOD could deploy powerful deep learning solutions to new, organization specific, media analysis challenges.

Starting with the hypothesis of whether or not the Department of Defense should procure and implement open-source algorithms and software libraries, we developed research questions necessary to explore this supposition. After selecting three previous defense-related theses and datasets, we developed experiments to apply an open-source machine learning framework to the same images to create a baseline for performance comparison. We captured timing metrics for the training and evaluation of the deep learning algorithms and related our measurements to realistic scenarios to show that open-source deep CNNs will function across several DOD environments. We observed that TensorFlow-based object detectors outperform or match previous algorithm performance, while requiring significantly less effort to apply them to each new problem. These observations provide empirical evidence supporting future DOD procurement and application of open-source machine learning software. Beyond our results, we argue that benefits of open-source deep learning libraries outweigh associated risk and that it is logical for the DOD to employ them. In illustrating TensorFlow's utility to the Department of Defense, we assert that the organization can benefit from the open-source innovations of industry and academia in the field of machine learning. Doing so affords the DOD an opportunity to increase efficiency and to keep pace with state-of-the-art advances in order to maintain a technological edge over its adversaries.

APPENDIX: Detailed Results

A.1 AK-47 Detector ROC Curves and Tables

A.1.1 Three Classification Scores Comparison

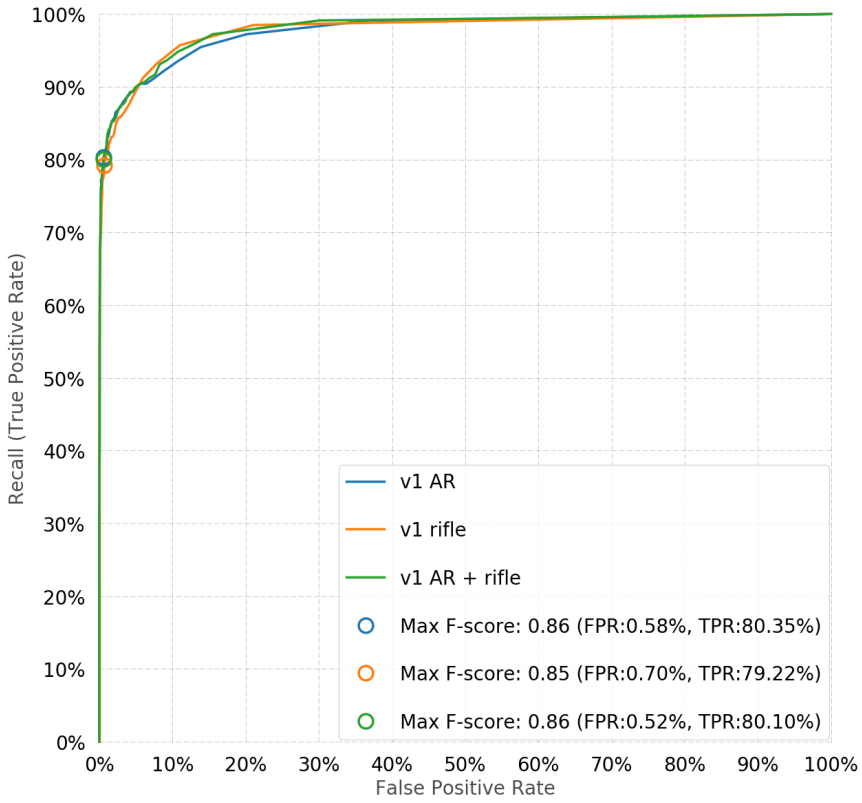


Figure A.1. Inception-v1 ROC curve for three category scores on all test sets.

Table A.1. Inception-v1 best F-score performance metrics all test sets.

Model	Thres.	Acc.	Prec.	FPR	Rec.	F-score
v1 AR	0.047	0.976	0.938	0.006	0.804	0.866
v1 rifle	0.022	0.973	0.925	0.007	0.792	0.853
v1 AR + rifle	0.072	0.976	0.944	0.005	0.801	0.866

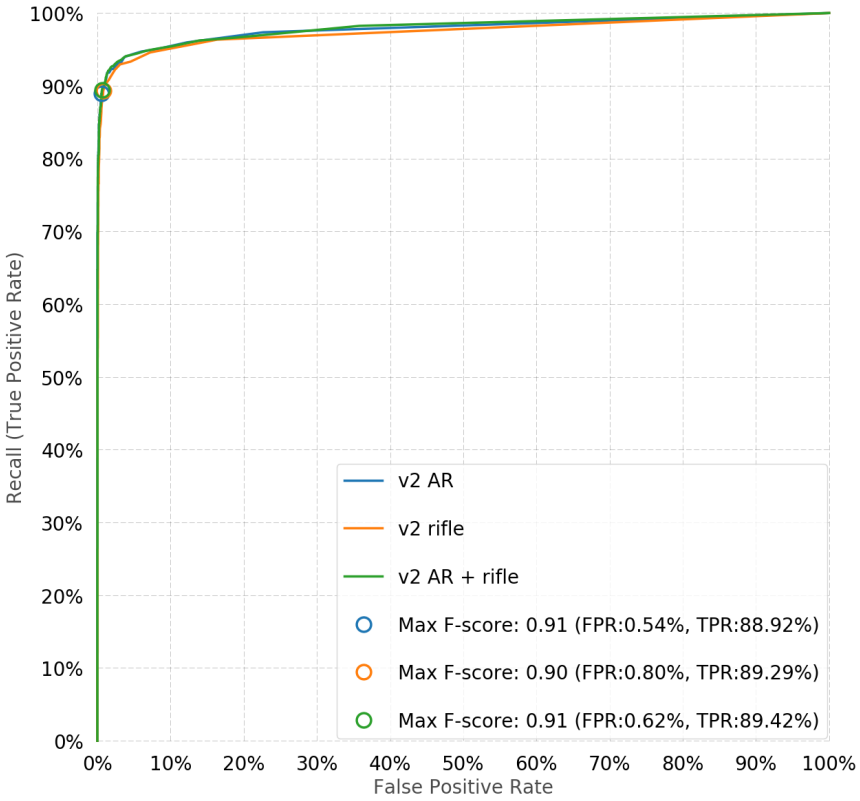


Figure A.2. Inception-v2 ROC curve for three category scores on all test sets.

Table A.2. Inception-v2 best F-score performance metrics all test sets.

Model	Thres.	Acc.	Prec.	FPR	Rec.	F-score
v2 AR	0.030	0.984	0.948	0.005	0.889	0.917
v2 rifle	0.011	0.982	0.924	0.008	0.893	0.908
v2 AR + rifle	0.040	0.984	0.940	0.006	0.894	0.917

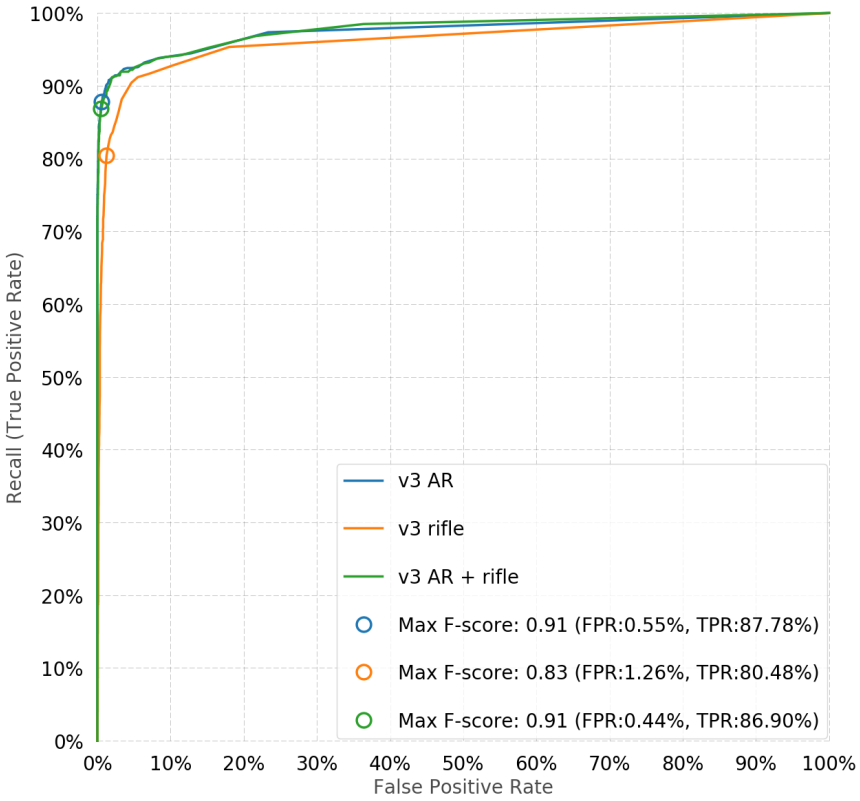


Figure A.3. Inception-v3 ROC curve for three category scores on all test sets.

Table A.3. Inception-v3 best F-score performance metrics all test sets.

Model	Thres.	Acc.	Prec.	FPR	Rec.	F-score
v3 AR	0.058	0.983	0.946	0.005	0.878	0.911
v3 rifle	0.016	0.969	0.874	0.013	0.805	0.838
v3 AR + rifle	0.108	0.983	0.956	0.004	0.869	0.910

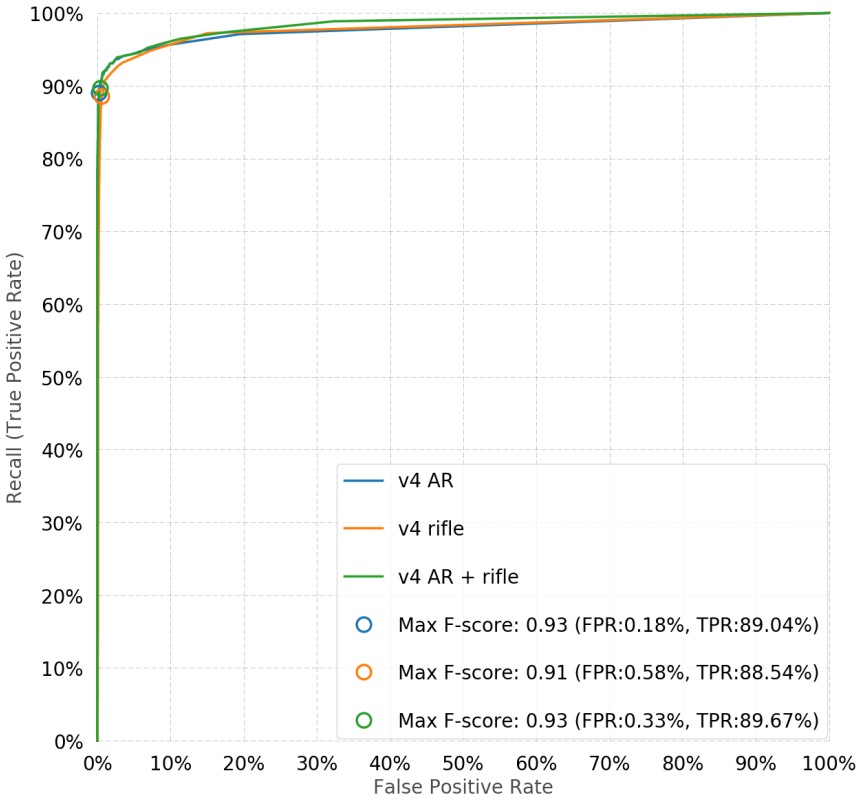


Figure A.4. Inception-v4 ROC curve for three category scores on all test sets.

Table A.4. Inception-v4 best F-score performance metrics all test sets.

Model	Thres.	Acc.	Prec.	FPR	Rec.	F-score
v4 AR	0.063	0.988	0.982	0.002	0.890	0.934
v4 rifle	0.019	0.984	0.944	0.006	0.885	0.914
v4 AR + rifle	0.077	0.987	0.967	0.003	0.897	0.931

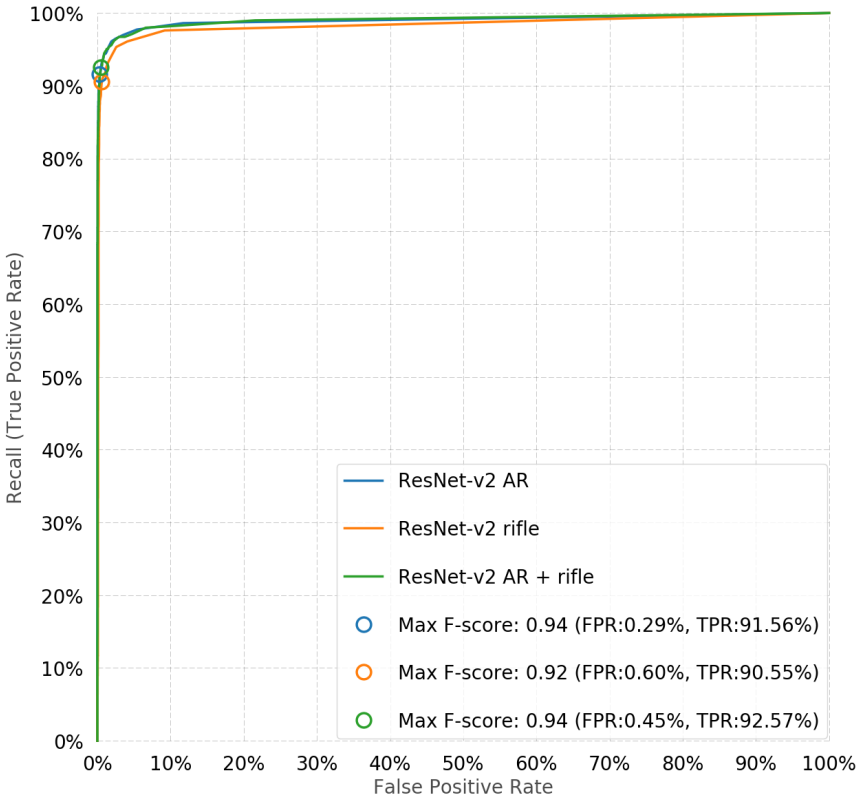


Figure A.5. Inception-ResNet-v2 ROC curve for three category scores on all test sets.

Table A.5. Inception-ResNet-v2 best F-score performance metrics all test sets.

Model	**Thres.**	**Acc.**	**Prec.**	**FPR**	**Rec.**	**F-score**
ResNet-v2 AR	0.036	0.989	0.972	0.003	0.916	0.943
ResNet-v2 rifle	0.011	0.985	0.942	0.006	0.906	0.924
ResNet-v2 AR + rifle	0.037	0.989	0.957	0.005	0.926	0.941

A.1.2 Jones' Test Set Performance Comparison

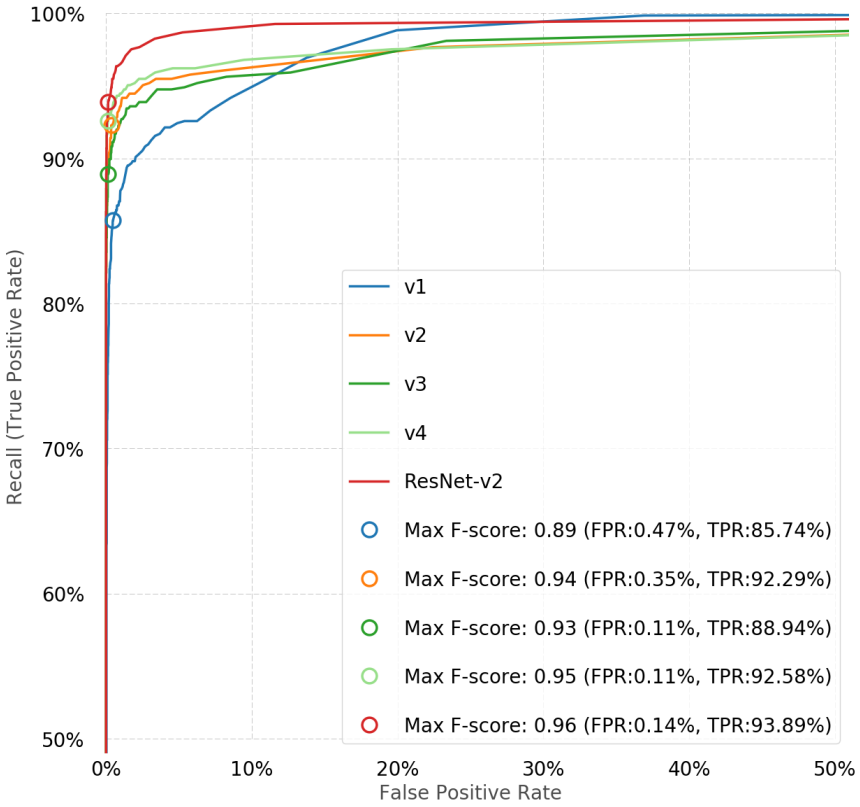


Figure A.6. AK-47 detector performance using the AR category for all Inception models on Jones' [15] test set (zoom in).

Table A.6. Inception models' best F-score performance metrics Jones test sets.

Model	Thres.	Acc.	Prec.	FPR	Rec.	F-score
v1	0.047	0.983	0.947	0.005	0.857	0.900
v2	0.030	0.990	0.962	0.004	0.923	0.942
v3	0.125	0.989	0.987	0.001	0.889	0.936
v4	0.053	0.992	0.988	0.001	0.926	0.956
ResNet-v2	0.036	0.993	0.985	0.001	0.939	0.961

A.1.3 All Test Set Performance Comparison

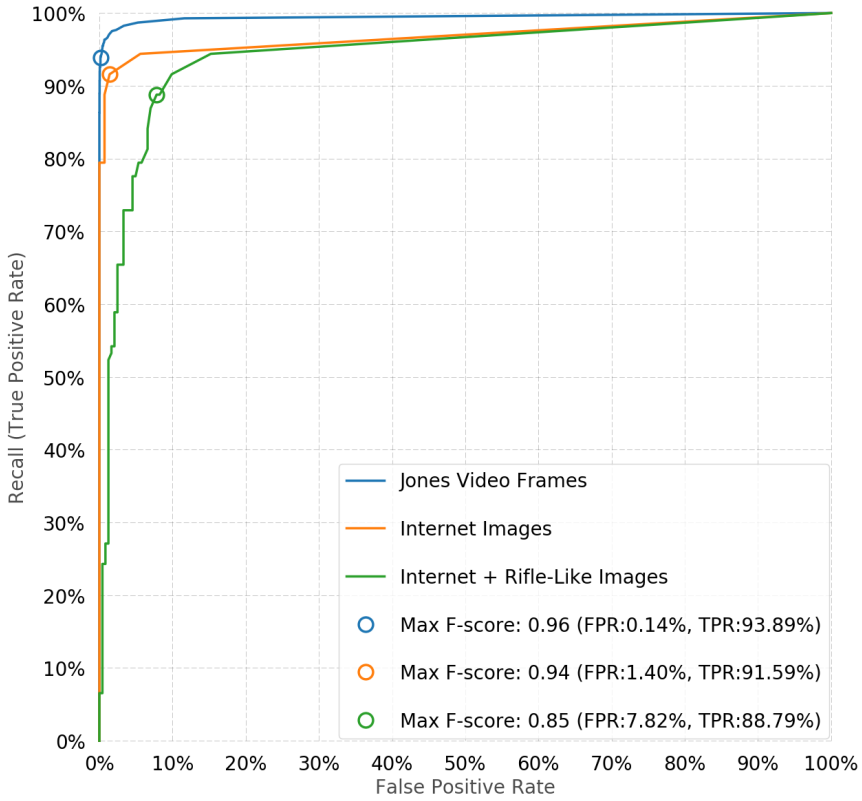


Figure A.7. Best AK-47 detector performance using the AR category for internet test sets.

Table A.7. Inception-ResNet-v2's best F-score performance metrics comparing Jones and internet test sets.

Image Set	Thresh.	Acc.	Prec.	FPR	Rec.	F-score
Jones Video Frames	0.036	0.993	0.985	0.001	0.939	0.961
Internet Images	0.002	0.956	0.980	0.014	0.916	0.947
Internet/Rifle-Like	0.004	0.911	0.833	0.078	0.888	0.860

A.2 Ship Detector ROC Curves and Tables

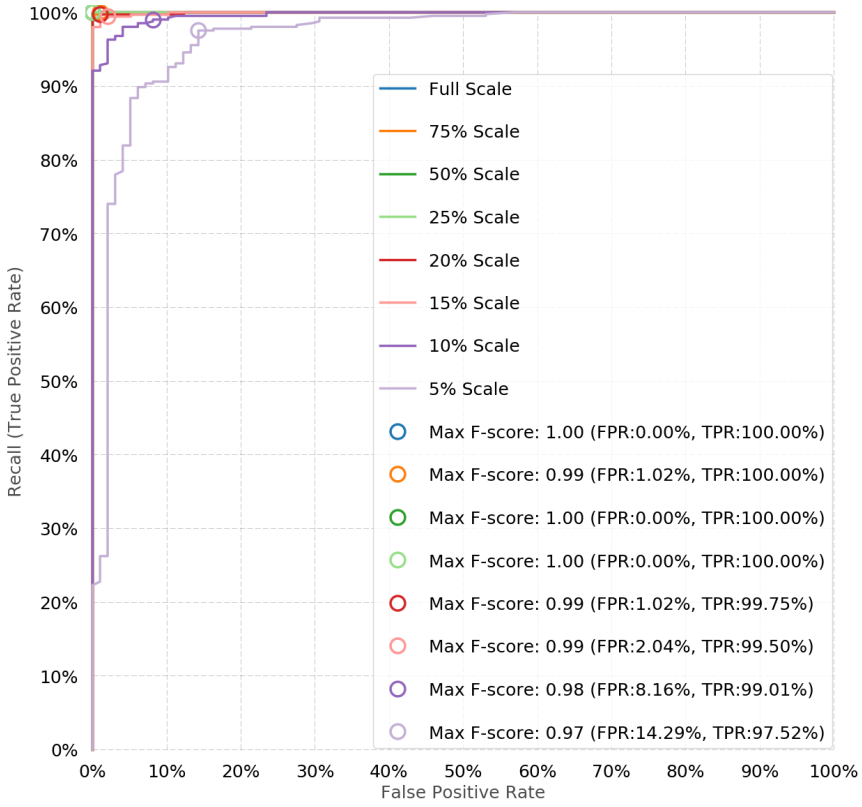


Figure A.8. Ship detector performance across all Camp's test set scales retrained with only Camp's training images.

Table A.8. Ship detector best F-score performance metrics on Camp's test set at all scales.

Image Set	Thresh.	Acc.	Prec.	FPR	Rec.	F-score
Full Scale	0.517	1.000	1.000	0.000	1.000	1.000
75% Scale	0.406	0.998	0.998	0.010	1.000	0.999
50% Scale	0.362	1.000	1.000	0.000	1.000	1.000
25% Scale	0.426	1.000	1.000	0.000	1.000	1.000
20% Scale	0.410	0.996	0.998	0.010	0.998	0.998
15% Scale	0.327	0.992	0.995	0.020	0.995	0.995
10% Scale	0.090	0.976	0.980	0.082	0.990	0.985
5% Scale	0.067	0.952	0.966	0.143	0.975	0.970

A.3 Screenshot Detector ROC Curves and Tables

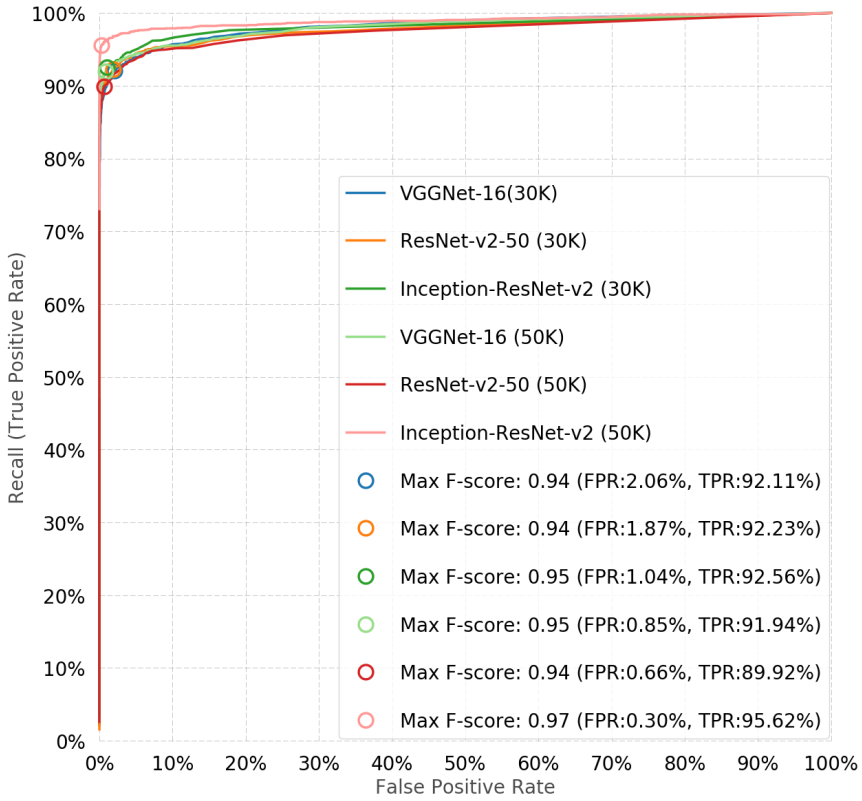


Figure A.9. Screenshot detector all models' performance on all Sharpe's [17] images.

Table A.9. Screenshot detector best F-score performance metrics on Sharpe's images all models.

Model	Thresh.	Acc.	Prec.	FPR	Rec.	F-score
VGGNet-16(30K)	0.097	0.956	0.967	0.021	0.921	0.944
ResNet-v2-50 (30K)	0.074	0.958	0.970	0.019	0.922	0.946
Inception-ResNet-v2 (30K)	0.055	0.964	0.983	0.010	0.926	0.954
VGGNet-16 (50K)	0.721	0.963	0.986	0.009	0.919	0.952
ResNet-v2-50 (50K)	0.056	0.956	0.989	0.007	0.899	0.942
Inception-ResNet-v2 (50K)	0.897	0.981	0.995	0.003	0.956	0.975

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. Available: http://dx.doi.org/10.1007/s11263-015-0816-y

[2] Y. Bengio, *Deep Learning of Representations: Looking Forward*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–37. Available: http://dx.doi.org/10.1007/978-3-642-39593-2_1

[3] R. Collobert *et al.*, "Torch7: A Matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[4] L. Buitinck *et al.*, "API design for machine learning software: Experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[5] R. Al-Rfou *et al.*, "Theano: A Python framework for fast computation of mathematical expressions," *CoRR*, vol. abs/1605.02688, Mar. 16 2016. Available: http://arxiv.org/abs/1605.02688

[6] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia* (MM '14). New York, NY: ACM, 2014, pp. 675–678. Available: http://doi.acm.org/10.1145/2647868.2654889

[7] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. Available: http://arxiv.org/abs/1603.04467

[8] T. Chen *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. Available: http://arxiv.org/abs/1512.01274

[9] D. Yu *et al.*, "An introduction to computational networks and the computational network toolkit," Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-2014-112, Oct. 2014. Available: https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/

[10] L. Levine and B. Novak, "Identifying acquisition patterns of failure using systems archetypes," in *2008 2nd Annual IEEE Systems Conference*, April 2008, pp. 1–6.

[11] D. Wennergren, *Clarifying Guidance Regarding Open Source Software (OSS)*, Memorandum, DOD Chief Information Officer, Washington, DC, Oct. 16, 2009, pp. 4–5. Available: http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf

[12] C. Szegedy *et al.*, "Rethinking the Inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. Available: http://arxiv.org/abs/1512.00567

[13] C. Szegedy *et al.*, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," *CoRR*, vol. abs/1602.07261, 2016. Available: http://arxiv.org/abs/1602.07261

[14] TensorFlow GitHub Repository. Contributors. [Online]. Available: https://github.com/tensorflow/tensorflow/graphs/contributors. Accessed Dec. 14, 2016.

[15] J. Jones, "Parts-based detection of ak-47s for forensic video analysis," M.S. thesis, Dept. Computer Science, Naval Postgraduate School, Monterey, California, 2010.

[16] D. Camp, "Evaluation of object detection algorithms for ship detection in the visible spectrum," M.S. thesis, Dept. Electrical Engineering, Naval Postgraduate School, Monterey, California, 2013.

[17] L. Sharpe, "Feature sets for screenshot detection," M.S. thesis, Dept. Computer Science, Naval Postgraduate School, Monterey, California, 2013.

[18] *The Defense Acquisition System*, DOD Directive 5000.1, Under Secretary of Defense (AT&L), Washington, DC, 2007, pp. 4–8.

[19] Defense Acquisition Guidebook. (2017, May 9). Defense Acquisition University. Fort Belvoir, VA. [Online]. Available: https://www.dau.mil/tools/dag

[20] *Operation of the Defense Acquisition System*, DoD Instruction 5000.02, Under Secretary of Defense (AT&L), Washington, DC, 2017, p. 6.

[21] J. R. Fox, *Defense Acquisition Reform, 1960-2009: An Elusive Goal*. Washington, DC: Government Printing Office, 2012, ch. 1, p. 14.

[22] D. Gouré. (2010, Sep. 8). Don't reform the acquisition system kill it. [Online]. Available: http://lexingtoninstitute.org/dont-reform-the-acquisition-system-kill-it/

[23] A. Oram, "Promoting open source software in government: The challenges of motivation and follow-through," *Journal of Information Technology & Politics*, vol. 8, no. 3, p. 242, June 3 2011. Available: http://dx.doi.org/10.1080/19331681.2011.592059

[24] G. Bradski. (2000, Nov. 1). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. [Online]. Available: http://www.drdobbs.com/open-source/the-opencv-library/184404319

[25] Wikimedia Commons. Category: Screenshots. [Online]. Available: https://commons.wikimedia.org/wiki/Commons:Screenshots. Lauren Sharpe accessed for her thesis in Apr. 2013.

[26] F.-F. Li and P. Perona, "A bayesian hierarchical model for learning natural scene categories," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Washington, DC: IEEE Computer Society, 2005, vol. 2, pp. 524–531. Available: http://dx.doi.org/10.1109/CVPR.2005.16

[27] F. Lundh *et al.* (2012). Python Imaging Library (PIL). [Online]. Available: http://www.pythonware.com/products/pil

[28] J. Demšar *et al.*, "Orange: Data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. Available: http://jmlr.org/papers/v14/demsar13a.html

[29] A. Krizhevsky *et al.*, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira *et al.*, Eds. Red Hook, NY: Curran Associates, Inc., 2012, pp. 1097–1105. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. Available: http://arxiv.org/abs/1409.1556

[31] K. He *et al.*, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. Available: http://arxiv.org/abs/1512.03385

[32] A. Karpathy, "Cs231n: Convolutional neural networks for visual recognition," *Online Course*, 2016. Available: http://cs231n.github.io/

[33] M. A. Nielsen. (2015). *Neural Networks and Deep Learning*. [Ebrary version]. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[34] N. Srivastava *et al.*, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. Available: http://jmlr.org/papers/v15/srivastava14a.html

[35] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. Available: http://arxiv.org/abs/1311.2901

[36] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, Oct. 2012. Available: http://doi.acm.org/10.1145/2347736.2347755

[37] S. R. Richter *et al.*, *Playing for Data: Ground Truth from Computer Games*. Cham: Springer International Publishing, 2016, pp. 102–118. Available: http://dx.doi.org/10.1007/978-3-319-46475-6_7

[38] J. Donahue *et al.*, "Decaf: A deep convolutional activation feature for generic visual recognition." in *Icml*, 2014, vol. 32, pp. 647–655.

[39] E. Yudkowsky, "Artificial Intelligence as a Positive and Negative Factor in Global Risk," in *Global Catastrophic Risks*, N. Bostrom and M. M. Ćirković, Eds. New York, NY: Oxford University Press, 2008, p. 303.

[40] T. Lin *et al.*, "Microsoft COCO: Common objects in context," *CoRR*, vol. abs/1405.0312, 2014. Available: http://arxiv.org/abs/1405.0312

[41] I. Krasin and T. Duerig. (2016, Sep. 30). Introducing the Open Images Dataset. [Online]. Available: https://research.googleblog.com/2016/09/introducing-open-images-dataset.html

[42] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[43] A. Alemi. (2016, Aug. 31). Improving Inception and image classification in TensorFlow. [Online]. Available: https://research.googleblog.com/2016/08/improving-inception-and-image.html

[44] TensorFlow GitHub Repository. TensorFlow-Slim Library. [Online]. Available: https://github.com/tensorflow/models/tree/master/slim. Accessed May 3, 2017.

[45] TensorFlow GitHub Repository. Pre-trained models. [Online]. Available: https://github.com/tensorflow/models/tree/master/slim#Pretrained. Accessed Dec 14, 2016.

[46] Image recognition. (n.d.). TensorFlow. [Online]. Available: https://www.tensorflow.org/versions/r0.11/tutorials/image_recognition/. Accessed May 1, 2017.

[47] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[48] T. Fawcett, “An introduction to ROC analysis,” *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[49] M. Sokolova *et al.*, “Beyond accuracy, F-score and ROC: A family of discriminant measures for performance evaluation,” in *Australasian Joint Conference on Artificial Intelligence*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1015–1021.

[50] How to retrain Inception’s final layer for new categories. (n.d.). TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/image_retraining. Accessed May 3, 2017.

[51] TensorFlow GitHub Repository. Inception-v3 retraining script (retrain.py). [Online]. Available: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/image_retraining. Accessed May 3, 2017.

[52] TensorFlow GitHub Repository. TensorBoard. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tensorboard. Accessed May 4, 2017.

[53] Wikimedia Commons. Category: Screencast videos. [Online]. Available: https://commons.wikimedia.org/wiki/Category:Screencast_videos. Accessed Mar. 28, 2017.

[54] J. Buchner. (n.d.). ImageHash Python Library. [Online]. Available: https://github.com/JohannesBuchner/imagehash. Accessed Aug. 4, 2016.

[55] N. Krawetz. (2016, May 5). Kind of like that. [Online]. Available: http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html

[56] TensorFlow GitHub Repository. TensorFlow-Slim Dataset Class. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/python/slim/data/dataset.py. Accessed May 6, 2017.

[57] Data IO (Python functions). (n.d.). TensorFlow. [Online]. Available: https://www.tensorflow.org/versions/r1.0/api_guides/python/python_io. Accessed May 6, 2017.

[58] TensorFlow GitHub Repository. Slim training script (train_image_classifier.py). [Online]. Available: https://github.com/tensorflow/models/blob/master/slim/train_image_classifier.py. Accessed May 6, 2017.

[59] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014. Available: http://arxiv.org/abs/1412.6980

[60] V. Library. (2007, July 16). Melissa shooting my AK again. [YouTube video]. Available: https://www.youtube.com/watch?v=KcvaAjP7QkY. Accessed May 24, 2017. Received permission to reprint via electronic correspondence.

[61] S. Riser. (n.d). Deployment of PALACE floats from voluntary observing ships (VOS) . . . [Online]. Available: http://flux.ocean.washington.edu/atlantic/info/about-deployments.html. Accessed May 22, 2017. Received permission to reprint via electronic correspondence.

[62] Ayrshire+ photographs: Ship on the horizon. (2006, November 23,). [Online]. Available: http://anhonestman.blogspot.com/2006/11/ship-on-horizon.html. Accessed May 22, 2017. Creative Commons Attribution-NonCommercial-NoDerivs 2.5 Generic License.

[63] yachtlightbluehotmail.com. (n.d). Yacht Light Blue foreign thoughts from abroad. [Online]. Available: http://www.yachtlightblue.co.uk/archives/sept_2006.html. Accessed May 22, 2017. Received permission to reprint via electronic correspondence.

[64] L. Nodé-Langlois. (2010, April 21,). Photoshop...nop ! [Online]. Available: http://lnl.aminus3.com/image/2010-04-24.html. Accessed May 25, 2017. Received permission to reprint via electronic correspondence.

[65] Wikimedia Commons: Kathe87. (2011, 23 June). File:Comandos para svn apache10.png. [Online]. Available: https://commons.wikimedia.org/wiki/File:Comandos_para_svn_apache10.png. Accessed May 23, 2017. Creative Commons Attribution-Share Alike 3.0 Unported License.

[66] Wikimedia Commons: Kathe87. (2011, 23 June). File:Comandos para svn apache21.png. [Online]. Available: https://commons.wikimedia.org/wiki/File:Comandos_para_svn_apache21.png. Accessed May 23, 2017. Creative Commons Attribution-Share Alike 3.0 Unported License.

[67] Wikimedia Commons: Mythtv Team. (2010, 6 April). File:MythTV-main menu.png. [Online]. Available: https://commons.wikimedia.org/wiki/File:MythTV-main_menu.png. Accessed May 23, 2017. GNU General Public License as published by the Free Software Foundation.

[68] Wikimedia Commons: Mdale. (2010, 3 September). File:Sequencer-drag-from-amw-to-sequence.jpg. [Online]. Available: https://commons.wikimedia.org/wiki/File:Sequencer-drag-from-amw-to-sequence.jpg. Accessed May 23, 2017. Creative Commons Attribution-Share Alike 3.0 Unported License.

[69] M. Meeker, "KPCB internet trends 2016," presented at Code Conference 2016, Rancho Palos Verdes, CA, June 1, 2016.

[70] Cloud Machine Learning Engine. Google Cloud Platform. [Online]. Available: https://cloud.google.com/ml-engine/. Accessed June 2, 2017.

[71] C. Severance. (n.d.). Programming for everybody (getting started with Python). [Online]. Available: https://www.coursera.org/learn/python. Accessed May 29, 2017.

[72] A. Ng. (n.d). Machine learning. [Online]. Available: https://www.coursera.org/learn/machine-learning. Accessed May 29, 2017.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California